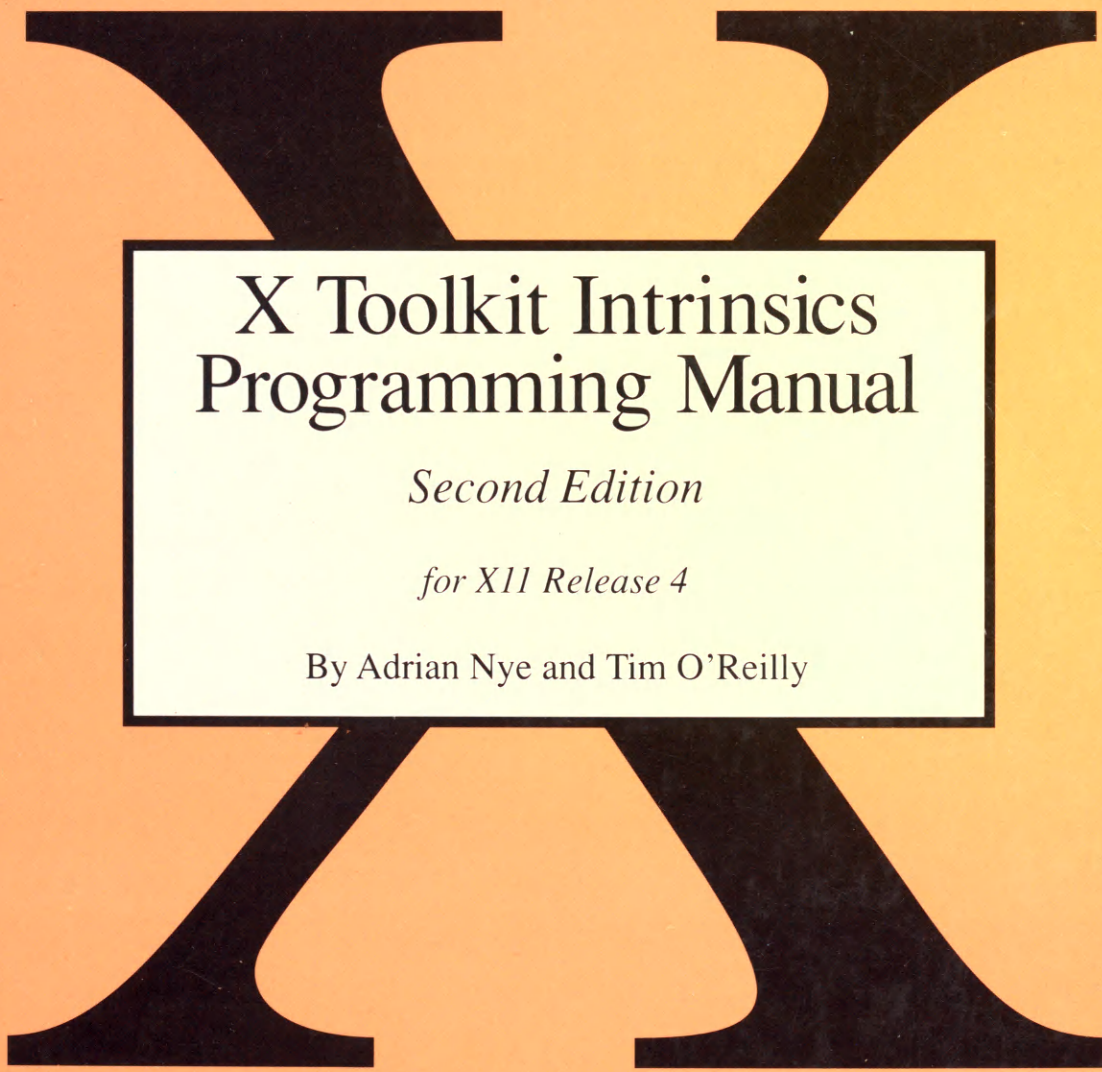
The Definitive Guides to the X Window System

Now With Complete Coverage of Release 4

Volume Four



X Toolkit Intrinsics Programming Manual

Second Edition

for X11 Release 4

By Adrian Nye and Tim O'Reilly

O'Reilly & Associates, Inc.

Volume Four

X Toolkit Intrinsics Programming Manual

Second Edition
for X11, Release 4

by Adrian Nye and Tim O'Reilly

O'Reilly & Associates, Inc.

Copyright © 1990 O'Reilly & Associates, Inc.

All Rights Reserved

The X Window System is a trademark of the Massachusetts Institute of Technology
UNIX is a registered trademark of AT&T
Macintosh is a registered trademark of Apple Computer, Inc.
MacPaint is a registered trademark of CLARIS Corporation.
OPEN LOOK is a trademark of AT&T
XView and SunView are trademarks of Sun Microsystems, Inc.
OSF/Motif is a trademark of the Open Software Foundation, Inc.
DECwindows is a trademark of Digital Equipment Corporation
Genera is a registered trademark of Symbolics, Inc.
Cedar is a trademark of Xerox, Inc.
Explorer is a trademark of Texas Instruments, Inc.
Tetris is a trademark of AcademySoft-ELORG
C++ is a trademark of AT&T

Revision and Printing History

First Printing January 1990.

Second Edition September 1990. Revised for R4.

Small Print

Portions of this document (chiefly the appendices) are based in part on *X Toolkit Intrinsics–C Language Interface*, by Joel McCormack, Paul Asente, and Ralph Swick, and *X Toolkit Athena Widgets–C Language Interface*, by Ralph Swick and Terry Weissman. both of which are copyright © 1985, 1986, 1987, 1988, 1989, 1990 the Massachusetts Institute of Technology, Cambridge, Massachusetts, and Digital Equipment Corporation, Maynard, Massachusetts.

We've used this material under the terms of its copyright, which grants free use, subject to the following conditions:

> "Permission to use, copy, modify and distribute this documentation (*i.e., the original MIT and DEC material*) for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of MIT or Digital not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. MIT and Digital make no representations about the suitability of the software described herein for any purpose. It is provided 'as is' without expressed or implied warranty."

Note, however, that those portions of this document that are based on the original X11 documentation and other source material have been significantly revised, and that all such revisions are copyright © 1990 O'Reilly & Associates, Inc. Inasmuch as the proprietary revisions can't be separated from the freely copyable MIT source material, the net result is that copying of this document is not allowed. Sorry for the doublespeak!

While every precaution has been taken in the preparation of this book, we assume no responsibility for errors or omissions. Neither do we assume any liability for damages resulting from the use of the information contained herein.

Volume 4: ISBN 0-937175-56-0 Set: ISBN 0-937175-58-7

The X Window System

The books in the X Window System Series are based in part on the original MIT X Window System documentation, but are far more comprehensive, easy to use, and are loaded with examples, tutorials and helpful hints. Over 20 major computer vendors recommend or license volumes in the series. In short, these are the definitive guides to the X Window System.

Volume 0:

X Protocol Reference Manual

A complete programmer's reference to the X Network Protocol, the language of communication between the X server and the X clients. 498 pages. $30.00.

Volumes 1 and 2:

Xlib Programming Manual
Xlib Reference Manual

Revised for Release 4. Complete guide and reference to programming with the X library (Xlib), the lowest level of programming interface to X. 672 and 792 pages. $60.00 for the set, or $34.95 each.

Volume 3:

X Window System User's Guide

Revised and enlarged for X11 Release 4. Describes window system concepts and the most common client applications available for X11. Includes complete explanation of the new window manager, *twm*, and a chapter on Motif. For experienced users, later chapters explain customizing the X environment. Useful with either Release 3 or Release 4. 749 pages. $30.00.

Volumes 4 and 5:

X Toolkit Intrinsics Programming Manual
X Toolkit Intrinsics Reference Manual

Complete guides to programming with the X Toolkit. The *Programming Manual* provides concepts and examples for using widgets and for the more complex task of writing new widgets. The *Reference Manual* provides reference pages for Xt functions, and Xt and Athena widgets. 582 and 545 pages. $55.00 for the set, or $30.00 each.

Volume 7:

XView Programming Manual

XView is an easy-to-use toolkit that is not just for Sun developers. It is available on MIT's R4 tape and System V Release 4, as well as being a part of Sun's Open Windows package. This manual provides complete information on XView, from concepts to creating applications to reference pages. 566 pages. $30.00.

The X Window System in a Nutshell

For the experienced X programmer, contains essential information from other volumes of the series in a boiled-down, quick reference format that makes it easy to find the answers needed most often. 380 pages. $24.95.

For orders or a free catalog of all our books, please contact us.

O'Reilly & Associates, Inc.

Creators and Publishers of Nutshell Handbooks
632 Petaluma Avenue, Sebastopol, CA 95472
email: uunet!ora!nuts · 1-800-338-6887 · 1-707-829-0515 · FAX 1-707-829-0104

Table of Contents

Page

Preface xxv

Summary of Contents xxv
Assumptions xxvi
Related Documents xxvii
How to Use This Manual xxvii
Font Conventions Used in This Manual xxx
Requests for Comments xxxi
Bulk Sales Information xxxi
Obtaining the X Window System Software xxxi
Example Programs xxxi
Acknowledgments xxxii

1 Introduction to the X Window System 3

1.1 The Server and Client 6
1.2 The Software Hierarchy 8
1.3 Event-driven Programming 10
1.4 The Window Manager 11
1.5 Extensions to X 12

2 Introduction to the X Toolkit 15

2.1 Programming with Widgets 15
2.1.1 Contents of a Widget Set 17
2.1.2 Widget Classes and Instances 18
2.1.3 Widget Configurability with Resources 23
2.1.4 Widget Independence 25
2.1.5 Widget-Application Interaction 27
2.1.6 Xt and Object-oriented Programming (OOP) 30
2.1.6.1 The Object 30
2.1.6.2 Methods 31
2.1.6.3 Messages 31
2.1.6.4 Encapsulation 31
2.2 Structure of X Toolkit Applications 32
2.3 A Simple X Toolkit Application 33
2.3.1 The Code 33
2.3.2 Compiling the Application 36

2.3.3 The Application-defaults File 38
2.3.4 To Hardcode or Not to Hardcode 40
2.4 Connecting Widgets to Application Code 41
2.4.1 Callbacks 42
2.4.2 Actions 45
2.4.2.1 The Actions Table 48
2.4.2.2 Format of an Action Procedure 48
2.4.2.3 The Translation Table 49
2.4.2.4 Hardcoding Translations 51
2.5 More About Resources 52
2.5.1 Setting and Getting Resources from the Application 52
2.5.1.1 Setting Resources with the Varargs Interfaces 53
2.5.1.2 Setting Resources with the ArgList Interfaces 54
2.5.1.3 Getting a Resource Value 56
2.5.2 Core Resources 57
2.5.3 Other Inherited Resources 59

3 More Techniques for Using Widgets 65

3.1 Using Composite Widgets 65
3.1.1 Setting Resources for an Instance Hierarchy 68
3.1.2 Geometry Management in Practice 69
3.2 Using Constraint Widgets 72
3.3 Using Pop Ups 75
3.4 More About Callbacks 81
3.4.1 Passing Data to Callback Functions 81
3.4.2 Callback Lists 84
3.5 Application Resources 85
3.5.1 The Application Data Structure 86
3.5.2 The Resource List 86
3.5.3 Getting the Resources 90
3.6 Command-line Options 91
3.6.1 Standard Command-line Options 91
3.6.2 Defining Your Own Command-line Options 93
3.7 Preventing User Customization of Widget Resources 97
3.7.1 Using the Varargs Interfaces 97
3.7.2 Using the Argument List Interfaces 99
3.7.2.1 Another Way to Set Arguments 100
3.7.2.2 Merging Argument Lists 102
3.8 More About Application Contexts 102

4 An Example Application 105

4.1 xbitmap1: Bitmap Editor Using a BitmapEdit Widget 106
4.1.1 Widget Public Functions 109
4.1.2 Application-defaults File 109

4.2 xbitmap2: Adding Scrollbars to Handle Large Bitmaps 109
4.2.1 Overriding Translations 114
4.2.2 Action Arguments in the Translation Table 114
4.2.3 The resize_thumbs Action 115
4.2.4 Scrollbar Callbacks 116
4.3 xbitmap3: Adding Graphics to Display the Bitmap 118
4.3.1 Graphics from the Application 121
4.3.2 Writing a Bitmap File 124
4.4 xbitmap4: Bitmap Editor Without a BitmapEdit Widget 125

5 Inside a Widget 137

5.1 Widget Source File Organization 138
5.2 The Private Header File—BitmapEdiP.h 138
5.2.1 Parts and Records 139
5.2.2 Class Part and Class Record 140
5.2.3 Instance Part and Instance Record 141
5.3 The Widget Implementation File—BitmapEdit.c 143
5.3.1 Obligatory Include Files 145
5.3.2 Defining the Resource List 145
5.3.3 The Translation Table and Actions Table 148
5.3.4 Declaring Methods 150
5.3.5 Initializing the Class Record 151
5.3.5.1 The Core Class Part 151
5.3.5.2 Initializing the Core Methods 153
5.3.6 Description of Core Methods 155
5.3.7 Packaging the Class Record for Application Use 157
5.3.8 A Sample Method 157
5.4 The Public Header File—BitmapEdit.h 160
5.5 The Process of Widget Writing 163
5.6 Summary of Conventions 164

6 Basic Widget Methods 169

6.1 The X Graphics Model Inside Widgets 170
6.2 The initialize Method 170
6.2.1 Creating GCs 172
6.3 The expose Method 175
6.4 The set_values Method 178
6.5 The resize Method 181
6.6 The query_geometry Method 184
6.7 The destroy Method 187
6.8 Actions in the Widget Framework 188

7 Events, Translations, and Accelerators 193

7.1 Translation Table Syntax 194
7.1.1 The Directive 195
7.1.2 Selecting the Events to Translate 195
7.1.3 Details in Keyboard Events 198
7.1.4 Details in Other Event Types 199
7.1.5 Modifiers 201
7.1.5.1 Physical Keys Used as Modifiers 202
7.1.5.2 Default Interpretation of the Modifier List 204
7.1.5.3 Prohibiting a Modifier 204
7.1.5.4 Requiring an Exact Match 205
7.1.5.5 Paying Attention to the Case of Keysyms 205
7.1.6 Event Sequences 206
7.1.6.1 Special Considerations Involving Motion Events 207
7.1.6.2 Modifiers and Event Sequences 207
7.1.6.3 Using Modifiers to Specify Button Event Sequences 208
7.1.6.4 Key Event Sequences 208
7.1.7 Interactions Between Translations 208
7.1.7.1 Translations in Multiple Resource Files 209
7.1.7.2 Order of Translations 209
7.1.7.3 Event Sequences Sharing Initial Events 210
7.1.7.4 Event Sequences Sharing Noninitial Events 210
7.2 Accelerators 211
7.2.1 Event Propagation 213
7.2.2 Installing Accelerators in Multiple Widgets 215
7.2.3 Defining the Accelerator Table in the Code 216
7.2.4 The display_accelerators Method 216

8 More Input Techniques 221

8.1 Event Handlers 222
8.1.1 Adding Event Handlers 224
8.1.2 Adding Nonmaskable Event Handlers 225
8.1.3 Removing Event Handlers 226
8.1.4 Adding Raw Event Handlers 227
8.2 Writing Routines That Use Specific Event Data 227
8.2.1 Event Types and Structure Names 229
8.3 File, Pipe, and Socket Input 231
8.3.1 Getting File Input 231
8.3.2 Getting Pipe Input 233
8.4 Timeouts 234
8.4.1 Visibility Interest 237
8.5 Work Procedures 239
8.6 Low-level Management of the Event Queue 241
8.6.1 XtPending and XtPeekEvent 241

8.6.2 Event Filters 242
8.6.3 Input Sensitivity 243

9 Resource Management and Type Conversion 247

9.1 Review of Resource Fundamentals 248
9.2 How the Resource Database Works 251
9.2.1 Form of Resource Specifications 252
9.2.2 Merging of Resource Files 253
9.2.3 The Language String 255
9.2.4 Fallback Resources 257
9.2.5 Resource Matching Algorithm 258
9.2.6 Resource Precedence Rules 260
9.3 Type Conversion 263
9.3.1 Conversions from XtRString 263
9.3.2 Other Built-in Type Conversions 266
9.3.3 Special Resource Defaults That Do Not Use Conversion 267
9.3.4 Registering Type Converters 269
9.3.4.1 Passing Arguments to a Type Converter 272
9.3.5 Explicitly Invoking a Converter 274
9.3.6 Writing a Type Converter 275
9.3.6.1 Defining the Default Value 277
9.4 Subparts and Subresources 278
9.4.1 The Hook Methods 279
9.4.2 Managing Subresources 279

10 Interclient Communications 283

10.1 Window Manager Interactions 284
10.1.1 Shell Subclasses 284
10.1.2 Setting Shell Resources 285
10.1.3 Screen Space 287
10.1.4 Input Model 288
10.1.5 Colormaps 290
10.1.6 Icons 292
10.1.7 Window Manager Decorations 293
10.2 Selections: Widget-To-Widget Communication 294
10.2.1 How Atomic Selection Works 295
10.2.2 Highlighting the Selected Data (Owner) 298
10.2.3 Making the Selection with XtOwnSelection (Owner) 303
10.2.4 Requesting the Selection (Requestor) 304
10.2.4.1 Possible Target Type Atoms 305
10.2.4.2 The Paste Action from BitmapEdit 307
10.2.5 Converting the Selection (Owner) 307
10.2.6 Finally Pasting the Selection (Requestor) 310
10.2.7 If the Selection is Lost (Owner) 312

10.2.8 When the Selection Transfer is Complete (Owner) 312
10.2.9 ICCCM Compliance 312
10.2.9.1 Xmu Atom Caching 314
10.2.9.2 Converting the Standard Selections 315
10.2.9.3 The Clipboard Selection 316
10.2.10 How Incremental Selection Works 317
10.2.11 Miscellaneous Selection Routines 318

11 Geometry Management 321

11.1 How Composite Management Works 322
11.1.1 Initial Geometry Negotiation 324
11.1.2 Basic Core Methods in Composite Widgets 328
11.1.3 Laying Out Child Widgets 330
11.1.4 The change_managed Method 332
11.1.5 XtQueryGeometry and the query_geometry Method 332
11.1.6 XtMakeGeometryRequest and the geometry_manager Method 333
11.1.7 The insert_child and delete_child Methods 335
11.2 How Constraint Management Works 336
11.3 Writing a Constraint Widget 336
11.3.1 The Core Resource List 336
11.3.2 The Constraint Resource List 337
11.3.3 Class Structure Initialization 339
11.3.4 The Constraint initialize Method 341
11.3.5 The class_part_init Method 342
11.3.6 The geometry_manager Method 343
11.3.7 The resize Method 347
11.3.8 The Core and Constraint set_values Methods 349
11.3.9 The change_managed Method 349
11.3.10 The query_geometry Method 350
11.3.11 Delaying Geometry Recalculation 351
11.4 Compound Widgets 351
11.5 Stacking Order 352

12 Menus, Gadgets, and Cascaded Pop Ups 357

12.1 Menu Styles and Implementation 359
12.1.1 How Menus are Popped Up 360
12.1.2 Menu Panes 361
12.2 Several Ways to Create and Use Menus 363
12.2.1 A Spring-Loaded Menu: Pointer Grabbing 363
12.2.2 A Drop-Down Menu 371
12.2.3 Cascaded Menus 374
12.2.4 Using the R4 SimpleMenu Widget 379
12.2.5 Delayed Popup Creation 383
12.3 About Dialog Boxes 384

12.4 Gadgets 385
12.4.1 Inside a Gadget 388
12.4.2 Private Header File 389
12.4.3 The Gadget Source File 390
12.4.4 The Public Header File 391
12.4.5 The Gadget Parent 391

13 Miscellaneous Toolkit Programming Techniques 399

13.1 Errors and Warnings 399
13.2 Objects 402
13.3 Macros For Getting Information 403
13.4 The Keyboard Focus and accept_focus Method 404
13.5 Keyboard Interpretation 406
13.6 Memory Allocation 408
13.7 Action Hooks and Calling Actions Directly 408
13.8 Xt Grabbing Functions 409
13.9 File Finding and Internationalization 409
13.10 Application Contexts 410
13.10.1 Multiple Application Contexts 411
13.10.2 Rewriting XtAppMainLoop for Multiple Application Contexts 412
13.10.3 Functions Used with Multiple Application Contexts 412
13.11 Multiple Top-level Shells 413
13.12 Connecting to Multiple Servers 413
13.13 Class Extension Structures 414

A OPEN LOOK and Motif 419

A.1 The AT&T OPEN LOOK Widgets 423
A.1.1 Application Controls 424
A.1.1.1 Command Buttons 424
A.1.1.2 Exclusive and Nonexclusive Settings 426
A.1.1.3 Analog Controls 428
A.1.2 Composite Widgets 428
A.1.2.1 Menus and Control Areas 428
A.1.2.2 General Purpose Composite Widgets 429
A.1.2.3 Scrollbars and Scrollable Windows 429
A.1.3 Pop Ups 431
A.1.4 Text Widgets 433
A.1.5 Drawing Areas 433
A.2 The OSF/Motif Widgets 433
A.2.1 Application Controls 436
A.2.1.1 Command Buttons 436
A.2.1.2 Analog Controls 436
A.2.2 Composite Widgets 437
A.2.2.1 Menus and Control Areas 437

A.2.2.2 General Purpose Composite Widgets 440
A.2.2.3 Scrollable Windows 441
A.2.3 Pop Ups 442
A.2.4 Text Widgets 443
A.2.5 Drawing Areas 443

B Specifying Fonts and Colors 447

B.1 Color Specification 447
B.1.1 Color Names 447
B.1.2 Hexadecimal Color Specification 448
B.1.2.1 The RGB Color Model 449
B.1.2.2 How Many Colors are Available? 450
B.2 Font Specification 452
B.2.1 Font Naming Conventions 453
B.2.2 Font Name Wildcarding 454
B.2.3 Font Name Aliasing 456
B.2.4 Making the Server Aware of Aliases 457
B.2.5 The fonts.dir Files 457
B.3 Window Geometry 458

C Naming Conventions 463

D Release Notes 467

D.1 Summary of New Functions and Procedure Types 467
D.2 Application Writing 470
D.2.1 New Function for Initializing an Application 470
D.2.2 Varargs Interfaces 470
D.2.2.1 Special Types for Varargs List Members 471
D.2.3 Loading the Resource Database 471
D.2.3.1 The Language String 471
D.2.3.2 New Built-In Application Resources 472
D.2.3.3 Fallback Resources 472
D.2.4 Parsing the Command Line 472
D.3 Changes Affecting Both Application and Widget Writing 473
D.3.1 Actions 473
D.3.1.1 Directly Invoking Actions 473
D.3.1.2 Action Hook Registration 473
D.3.1.3 Obtaining Event Information in an Action Procedure 473
D.3.1.4 Registering Button and Key Grabs For Actions 474
D.3.1.5 Action Functions and Unrealized Widgets 474
D.3.2 Reserved Names for Resources and Actions 474
D.3.3 Resource String Identifier Convention 474

D.3.4 Resource File Format 474
D.3.5 Modifier Names 475
D.3.6 Determining Specification Revision Level 475
D.3.7 Implementation-specific Datatype Definitions 475
D.3.8 Event Handlers 476
D.3.9 Error and Warning Handlers 476
D.4 Changes Affecting Widget Writing 476
D.4.1 New Name for Core Class 476
D.4.2 File Content Changes 477
D.4.2.1 Changes to Private .h Files 477
D.4.2.2 Changes to .c Files 477
D.4.3 Class Extension Structures 477
D.4.3.1 Composite Class Extension Structure 478
D.4.3.2 Constraint Class Extension Structure 479
D.4.3.3 New Constraint get_values_hook method 479
D.4.3.4 Shell Class Extension Structure 479
D.4.4 New XtNvisual Resource 480
D.4.5 Methods 480
D.4.6 Mapping Order of Realized Child Widgets 481
D.4.7 Callback Lists 481
D.4.7.1 XtNunrealizeCallback 481
D.4.7.2 Internal Format of Callback Lists 481
D.4.7.3 New Routine to Call Callback List 482
D.4.8 Resource Lists 482
D.4.8.1 Base Offsets in Resource List 482
D.4.8.2 XtProcedureArg 482
D.4.8.3 Default Resource Values 482
D.4.8.4 Getting a Constraint Resource List 482
D.4.9 Converters 483
D.4.9.1 New Xt Resource Representations 483
D.4.9.2 New Xt Resource Converters 483
D.4.9.3 New Fn for Issuing Conversion Warnings 484
D.4.9.4 New Resource Converters Interface 484
D.4.10 Keyboard Handling 486
D.4.10.1 KeyCode-to-KeySym Conversions 486
D.4.10.2 KeySym-to-KeyCode Conversions 486
D.4.11 Selections 486
D.4.12 Objects 486
D.4.12.1 The Gadget Parent 487
D.4.12.2 Xt Functions that take any Object 488
D.4.12.3 Xt Functions that take any RectObj 489
D.4.12.4 Xt Functions that take any Core 489
D.4.12.5 Xt Functions that take any Composite 490
D.4.12.6 Xt Functions that take any Shell or a Subclass 490
D.4.12.7 Macros for Getting Object Information 490
D.4.13 Miscellaneous 491
D.4.13.1 XtNameToWidget 491
D.4.13.2 New Exposure Compression Controls 491

D.4.13.3 Requesting Key and Button Grabs 492
D.4.13.4 New Macros for Widget Superclass Determination 493
D.4.13.5 Macros and Procedure Equivalents 493
D.4.13.6 Finding File Names 493
D.4.13.7 Improved Multi-Display Support in Xmu 493

E The xbitmap Application 497

E.1 The BitmapEdit Widget 497
E.2 The BitmapEdiP.h Private Header File 510
E.3 The BitmapEdit.h Public Header File 511
E.4 xbitmap8 513

F Sources of Additional Information 521

F.1 Getting the X Software 521
F.1.1 Bug Fixes 522
F.1.1.1 Notes 524
F.1.1.2 Fairness 524
F.2 Netnews 525
F.3 Training and Consulting 525
F.3.1 Phone Support 526
F.4 The X Consortium 526
F.5 Finding Out for Yourself 527

Glossary 531

Master Index 553

Figures

Page

1-1 An X application, and an application on a traditional text terminal 4
1-2 A three-button mouse directing the pointer to select a menu item 4
1-3 Screen layout of a typical user's X Window System 5
1-4 Applications can run on any system across the network 7
1-5 The software architecture of Xt Intrinsics-based applications 9
2-1 A widget-based application 16
2-2 Inheritance among the Athena widgets 20
2-3 An Athena Paned widget 21
2-4 An Athena Dialog widget 22
2-5 A Viewport widget 22
2-6 Several Label widgets configured using resources 24
2-7 Widgets operate independently of the application 26
2-8 XtSetValues lets the application set how a widget will operate itself 27
2-9 Application registers a function with Xt during startup phase 28
2-10 Xt calls the function during the event-loop phase in response to an occurrence 29
2-11 xhello: appearance on screen 33
2-12 The appearance of xgoodbye when the pointer is in the window 43
2-13 xfarewell run with new resource settings 61
3-1 xbox1: appearance on the screen 66
3-2 Two configurations of xbox1 69
3-3 Results upon resize of Scrollbar widgets inside a Box widget 71
3-4 The xmh application and its instance hierarchy 72
3-5 Effect of the Form XtNfromHoriz resource 74
3-6 xbox3: popping up a Dialog widget 76
3-7 The widget instance and X window hierarchies are different only for pop ups 80
4-1 xbitmap1: how it looks on the screen 106
4-2 xbitmap2: scrollbars added 110
4-3 The instance hierarchy of xbitmap2 112
4-4 xbitmap3: true-scale normal and reverse bitmaps added 118
4-5 Application draws into Pixmap and copies it to widget window 119
5-1 The class hierarchy of the BitmapEdit widget (with other classes shown dotted) 139
5-2 Order of code in widget .c file 144
6-1 compress_exposure: 2 rectangles if XtExposeNoCompress; bounding box and region if XtExposeCompressSeries or XtExposeCompressMultiple 177
7-1 xbox1: two Command widgets in a Box 212
7-2 Key event propagation in xbox 214
8-1 xtetris in play 235
9-1 Suggested resource setting search path on UNIX-based systems 255
9-2 The quit widget in a sample widget hierarchy 259
9-3 Steps in matching resource entries for one resource of widget being created 264
10-1 An icon with and without an icon pixmap 292
10-2 The process of selection transfer 297
11-1 Role of border width in widget geometry 323

11-2 Initial geometry negotiation, assuming sufficient shell space 325
11-3 Initial geometry negotiation, if resizing is necessary 326
11-4 A ScrollBox widget at two different sizes 326
11-5 Geometry negotiation by the set_values_almost method 334
12-1 Command widgets in a button box, and the same commands as a menu 358
12-2 Menus from the OPEN LOOK and Motif widget sets 362
12-3 xmenu1: application with spring-loaded pop-up menu 366
12-4 xmenu2: a drop-down menu 371
12-5 xmenu5: cascaded spring-loaded menus 375
12-6 xmenu7: a menu using the Athena SimpleMenu widget 380
12-7 Class hierarchy derivation of Core 387
13-1 Using XtCallAcceptFocus to set the keyboard focus to a child 406
A-1 An OPEN LOOK application 423
A-2 Class inheritance hierarchy of the AT&T OPEN LOOK widgets 425
A-3 An OPEN LOOK ControlArea with OblongButton and ButtonStack widgets 425
A-4 OPEN LOOK RectButtons controlled by Exclusives and Nonexclusives widgets 427
A-5 An OPEN LOOK CheckBox widget 428
A-6 An OPEN LOOK Slider widget 428
A-7 An OPEN LOOK Scrollbar 429
A-8 An OPEN LOOK ScrollingList widget 430
A-9 An OPEN LOOK Notice 431
A-10 An OPEN LOOK PopupWindow 432
A-11 An OPEN LOOK Help window 432
A-12 Look and feel of a Motif application 434
A-13 Class inheritance hierarchy of the Motif widget set 435
A-14 Motif DrawnButton and PushButton widgets 436
A-15 A Motif RowColumn widget configured as a Menu Bar 438
A-16 A Motif RowColumn widget configured with four Radio Boxes 438
A-17 A Motif RowColumn widget configured as a drop-down menu 439
A-18 A Motif Form widget and children 440
A-19 A Motif ScrollBar 441
A-20 A Motif SelectionBox 442
B-1 Multiple planes used to index a colormap 451
B-2 A font name 453

Examples

Page

2-2 XHello: the application-defaults file 39
2-3 xgoodbye.c: complete code 43
2-4 XGoodbye: the application-defaults file 44
2-5 xfarewell.c: complete code 46
2-6 An XtActionProc with widget and event arguments 48
2-7 XFarewell: the application-defaults file 50
2-8 Code Fragment: specifying translations in the program 52
2-9 Using XtVaSetValues to set a widget resource 54
2-10 Using XtSetValues to set a widget resource 54
2-11 Code fragment to set multiple resources of a widget 55
2-12 Code fragment to get a widget resource using XtVaGetValues 56
2-13 Code fragment to get a widget resource using XtGetValues 57
2-14 Alternate resource settings for xfarewell 60
3-1 xbox1.c: complete code 66
3-2 XBox1: application-defaults file 68
3-3 XBox2: application-defaults file 74
3-4 xbox3: application-defaults file 75
3-5 Creating a pop-up dialog box 76
3-6 Passing a single value to a callback function 82
3-7 Passing a pointer to a structure to a callback function 82
3-8 ANSI non-conformant callback function definition 83
3-9 Initializing a callback list 85
3-10 xbitmap: getting application resources 86
3-11 The resource list for xbitmap 87
3-12 Calling XtGetApplicationResources and checking values 90
3-13 xbitmap: specifying command-line options 93
3-14 The R4 varargs interface to creating a widget 97
3-15 An argument list 99
3-16 Using an argument list in widget creation 99
3-17 Setting the argument list with XtSetArg 100
3-18 Incorrectly setting the argument list with XtSetArg 101
4-1 xbitmap1: complete code 107
4-2 XBitmap1: application-defaults file 109
4-3 xbitmap2: adding scrollbars 112
4-4 xbitmap2: resize_thumbs routine 115
4-5 xbitmap2: scrolling callback functions 116
4-6 xbitmap3: implementing small pixmaps by drawing into Core widgets 120
4-7 xbitmap3: the cell_toggled routine 122
4-8 xbitmap3: the Redraw_small_picture routine 123
4-9 xbitmap3: writing a bitmap file 124
4-10 xbitmap4: implementing the bitmap editor from the application 126
5-1 BitmapEdiP.h: the class part and class record 140
5-2 BitmapEdiP.h: the instance part and instance record 141

5-3 BitmapEdiP.h: declaring the external class record 143
5-4 BitmapEdit.c: include files 145
5-5 BitmapEdit's resource list 145
5-6 Setting the resource list into the class structure 148
5-7 The default translation table and the actions table 149
5-8 Translations in the Core class record 149
5-9 BitmapEdit.c: function type declarations 150
5-10 BitmapEdit.c: initialization of Core class record 151
5-11 BitmapEdit.c: declaring the class record pointer 157
5-12 BitmapEdit.c: inheriting a self-contained method 158
5-13 Inheriting by invoking the superclass method from a widget method 159
5-14 BitmapEdit.h: incidental declarations 161
6-1 The initialize method 171
6-2 Creating GCs from the initialize method 173
6-3 The expose method 175
6-4 The set_values method 178
6-5 BitmapEdit: the resize method 182
6-6 The XtWidgetGeometry structure 184
6-7 BitmapEdit: the query_geometry method 186
6-8 The destroy method 187
6-9 BitmapEdit: action routines 188
7-1 Installing accelerators in an application 212
7-2 Specifying the XtNaccelerators resource from the application-defaults file 212
7-3 Conflicting translations and accelerators 214
7-4 Installing accelerators from both command widgets 215
7-5 Accelerators resource settings for two widgets 215
8-1 Registering an event handler, and the handler function itself 224
8-2 Adding a nonmaskable event handler 226
8-3 Casting the event structure by declaring action routine arguments 228
8-4 Handling multiple event types in an action routine 228
8-5 Getting file input with XtAppAddInput 232
8-6 Reading stdin from an Xt application 234
8-7 xtetris: registering a timeout and the timeout function 235
8-8 xtetris: calling XtRemoveTimeOut 237
8-9 Registering an Xt Work Procedure 239
8-10 A work procedure to create a pop-up widget 240
8-11 Skeleton of a custom main loop 241
9-1 Two resources defined by the Athena Label widget 248
9-2 An app-defaults file for the Spanish language 257
9-3 Setting fallback resources in XtAppInitialize 257
9-4 A sample merged resource database 258
9-5 Sample resource database with eliminated entries and components 260
9-6 Resource database after final elimination of entries and components 260
9-7 Resource database finalists in original form 261
9-8 A resource definition converting an int to a pixmap 267
9-9 A resource definition using XtRImmediate 267
9-10 A resource definition using XtRCallProc 268
9-11 An example of an XtResourceDefaultProc 268

9-12 Registering a type converter 270
9-13 Adding a converter with arguments 272
9-14 Manually invoking a type converter 274
9-15 An R4 XtRString to XtREdgeType type converter 276
9-16 Testing for a special-case default value 278
9-17 Simplified get_values_hook method of the AsciiSrc subpart of the Text widget 279
10-1 Setting the XtNinput resource of a Shell widget 290
10-2 Creating an icon pixmap, and setting XtNiconPixmap 292
10-3 BitmapEdit: actions that highlight selection 299
10-4 BitmapEdit: getting the atom for a widget-specific target type 306
10-5 BitmapEdit: action to paste a selection 307
10-6 BitmapEdit: converting the selection value 308
10-7 BitmapEdit: pasting selection in requestor_callback function 310
10-8 BitmapEdit: the lose_ownership_proc 312
10-9 BitmapEdit: initializing Xmu's atom caching mechanism in the initialize method . 314
10-10 BitmapEdit: converting standard targets in the convert_proc 315
11-1 ScrollBox: the set_values method 328
11-2 ScrollBox: the resize method 329
11-3 ScrollBox: the query_geometry method 329
11-4 ScrollBox: private routine to lay out child widgets 330
11-5 Form: the Core resource list 336
11-6 Form: constraint resource list 337
11-7 Form: constraint data structure 338
11-8 Form: class structure initialization 339
11-9 Form: the Constraint initialize method 341
11-10 Form: the class_part_init method 342
11-11 Form: the geometry_manager method 343
11-12 Form: private functions: RefigureLocations and the layout method 344
11-13 Form: the LayoutChild private function 346
11-14 Form: the resize method 347
11-15 Form: the change_managed method 350
11-16 Form: the query_geometry method 350
11-17 Form: the public function for delaying calls to change_managed 351
12-1 xmenu1: complete code 366
12-2 XMenu1: the application-defaults file 369
12-3 xmenu2: code to place drop-down menu 372
12-4 XMenu2: translation portion of the application-defaults file 374
12-5 xmenu5: actions that place, pop up, and pop down main menus and submenus 376
12-6 XMenu5: translation portion of application-defaults file 378
12-7 xmenu7: using the SimpleMenu widget and its children 380
12-8 XMenu7: application-defaults file 382
12-9 Sme gadget: class part and complete class structure declaration 389
12-10 Sme gadget: instance part and complete instance structure declaration 390
12-11 SimpleMenu: expose method calling gadget children's expose methods 392
12-12 SimpleMenu: resize method 393
12-13 SimpleMenu: the Notify action routine 394
12-14 SimpleMenu.c: Setting accepts_objects in the Composite extension structure 394
13-1 How to invoke XtAppErrorMsg (from AsciiSrc.c) 401

13-2 Common fields in a class extension record 414
B-1 Sample fonts.alias file 456
B-2 fonts.dir file in /usr/lib/X11/fonts/100dpi 458
E-1 BitmapEdit: complete widget code 497
E-2 BitmapEdiP.h: complete private header file 510
E-3 BitmapEdit.h: complete public header file 511
E-4 xbitmap8: complete application code 513

Tables

		Page
2-1	Functions that Set Resources: arglist and varargs Counterparts	53
2-2	Core Resources	57
2-3	Label Resources	60
3-1	Resource Type Strings	89
3-2	Standard Command-line Parameters	92
3-3	XrmOptionKind: Command-line Option Style Constants	95
5-1	Resource List Constant Conventions	147
5-2	Inheritance Style of Various Methods	155
5-3	Summary of Xt Structure Name Conventions	164
6-1	XtWidgetGeometry request_mode Symbols	184
6-2	XtWidgetGeometry stack_mode Symbols	185
7-1	Event Type Abbreviations in Translation Tables	196
7-2	Notify Mode Values for Enter, Leave, and Focus Events	200
7-3	Atom Details for Various Events	200
7-4	Event Structure Fields Used as Translation Table Hints	201
7-5	Modifiers Used in Translation Tables	202
7-6	Modifier Symbols	204
8-1	Event Masks and Event Types	223
8-2	Event Types and Event Structures	230
8-3	Other Input Source Masks	231
9-1	Resource Type Strings	250
9-2	Built-in Type Converters from XtRString	265
9-3	Other Built-in Converters	266
9-4	Xmu Converters	269
10-1	Shell Resources	285
10-2	Target Types Suggested in ICCCM	313
13-1	Xt Error and Warning Message Utilities	401
13-2	Xt Routines That Use Default and Explicit Application Contexts	410
A-1	Comparison of Athena, OPEN LOOK, and Motif Widgets	420
B-1	Geometry Specification: x and y Offsets	459
D-1	ArgList Functions and varargs Counterparts	470
D-2	New Naming for Core Class Symbols and Variables	476
D-3	New Representation Types	483
D-4	Flags for compress_exposure field of Core Class Structure	492
D-5	Optional OR Flags for compress_exposure Field of Core Class Structure	491

Preface

By convention, a preface introduces the book itself, while the introduction starts in on the subject matter. You should read through the preface to get an idea of how the book is organized, the conventions it follows, and so on.

In This Chapter:

Summary of Contents xxv
Assumptions xxvi
Related Documents xxvii
How to Use This Manual xxvii
Font Conventions Used in This Manual xxx
Requests for Comments xxxi
Bulk Sales Information xxxi
Obtaining the X Window System Software xxxi
Example Programs xxxi
Acknowledgments xxxii

Preface

The Xt Intrinsics library consists of routines for building and using widgets. A *widget* is a pre-built user-interface component. Applications are written using the Xt Intrinsics library, together with a widget set library, to simplify the development of X Window System applications. This combination of Xt and any widget set is collectively called the X Toolkit. This book describes how to build applications using the Xt Intrinsics and any widget set. It provides a complete tutorial with programming examples on how to program with existing widgets and how to write your own.

There are several widget sets provided by system vendors to implement their particular user-interface styles. The two most widely used are OSF's Motif, and AT&T's OPEN LOOK widget set. The X distribution from MIT (where X itself is developed) comes with the Xaw widget set, which provides a small number of widgets that can be used to write simple application programs. Xaw was developed by MIT's Project Athena, and the acronym Xaw stands for "Athena Widgets." Xaw was designed as a simple demonstration and test of the Intrinsics—not as a complete set of widgets for writing commercial applications.

This book uses the Athena widgets to demonstrate how to use existing widgets, but what it says is equally applicable to and provides a good introduction to programming with any other widget set based on Xt. It is based on Release 4 of the Intrinsics and the Athena widgets.

Xt is written using Xlib, the lowest level C-language interface to the X Window System. Both the Xt Intrinsics and Xlib are required by the X standard (established by the X Consortium) on any system that allows programming of X applications in C.

Summary of Contents

The discussion of the X Toolkit is divided into two volumes, Volumes Four and Five of the X Window System Series available from O'Reilly & Associates, Inc.

This is Volume Four, *X Toolkit Intrinsics Programming Manual*. It provides an explanation of the X Toolkit, including tutorial material and numerous programming examples. Arranged by task or topic, each chapter brings together a group of Xt functions, describes the conceptual foundation on which they are based, and illustrates how they are most often used in writing applications. This volume is structured to be useful as a tutorial and also as a task-oriented reference.

Volume Five, the *X Toolkit Intrinsics Reference Manual*, includes reference pages for each of the Xt functions, as well as for the widget classes defined by Xt, organized alphabetically for ease of reference; a permuted index; and numerous appendices and quick reference aids.

The two volumes are designed to be used together. To get the most out of the examples in Volume Four, you will need the exact calling sequences of each function from Volume Five. To understand fully how to use each of the functions described in Volume Five, all but the most experienced Toolkit "hacker" will need the explanation and examples in Volume Four.

Both volumes include material from the original Toolkit documentation provided by MIT, though in Volume Four this material is mostly limited to the appendices. We have done our best to incorporate all the useful information from the MIT documentation, to reorganize and present it in a more useful form, and to supplement it with conceptual material, tutorials, reference aids, and examples. In other words, this manual is not only a replacement but is a superset of the MIT documentation.

Those of you familiar with the MIT documentation will recognize that each reference page in Volume Five includes the detailed description of the routine found in *X Toolkit Intrinsics—C Language Interface*, plus in many cases additional text that clarifies ambiguities and describes the context in which the routine would be used. We have also added cross-references to related reference pages and to where additional information can be found in Volume Four.

Assumptions

This book makes no assumptions about the reader's knowledge of object-oriented programming or the X Window System. Readers should be proficient in the C programming language, although examples are provided for infrequently used features of the language that are necessary or useful when programming with the X Toolkit. In addition, general familiarity with the principles of raster graphics will be helpful.

However, even though the Toolkit is intended to hide the low-level X interface provided by Xlib, there are times in writing applications or widgets when Xlib functions will be necessary because no Xt feature exists to do the same thing. This book describes the most common occasions for using Xlib, but does not provide a reference to the particular functions involved. Additional documentation on Xlib, such as that provided by Volume One, *Xlib Programming Manual*, and Volume Two, *Xlib Reference Manual*, will be indispensable.

Related Documents

Seven other books on the X Window System are available from O'Reilly & Associates, Inc.:

Volume Zero	*X Protocol Reference Manual*
Volume One	*Xlib Programming Manual*
Volume Two	*Xlib Reference Manual*
Volume Three	*X Window System User's Guide*
Volume Five	*X Toolkit Intrinsics Reference Manual*
Volume Seven	*XView Programming Manual*
Quick Reference	*The X Window System in a Nutshell*

The following documents are included on the X11 source tape:

X Toolkit Intrinsics—C Language Interface, by Joel McCormack, Paul Asente, and Ralph Swick
X Toolkit Athena Widgets—C Language Interface, by Ralph Swick and Terry Weissman
Xlib—C Language X Interface, by Jim Gettys, Ron Newman, and Robert Scheifler

The following Nutshell Handbooks published by O'Reilly and Associates, Inc. are useful when programming in C:

Checking C Programs with lint, by Ian Darwin
Managing Projects with make, by Steve Talbott
Using C on the UNIX System, by Dave Curry

The following is the classic introduction to C programming:

The C Programming Language, by B. W. Kernighan and D. M. Ritchie

How to Use This Manual

Volume Four explains both application programming with widgets and widget programming (the design and coding of new widgets).

The first four chapters treat widgets largely as "black boxes," which is appropriate considering the object-oriented philosophy of the Toolkit. These chapters also provide an overview of many elements of the X Toolkit, and so are appropriate for all readers.

Chapter 1 *Introduction to the X Window System*, provides a discussion of the context in which X programs operate. Programmers who are comfortable programming with Xlib can skip Chapter 1.

Chapter 2 *Introduction to the X Toolkit*, describes the conceptual foundations underlying Toolkit programming, and shows how to write simple programs that use widgets from existing widget sets. It introduces such fundamental Toolkit

programming concepts as resources, the Translation Manager, callbacks, and actions.

Chapter 3 *More Techniques for Using Widgets*, describes how to use some of the more complex widgets found in applications, including composite widgets, constraint widgets, and pop ups. It also describes how to define application resources and command-line options, and how to hardcode the value of widget resources when you create a widget. Finally, it describes how to create multiple top-level windows, and how to use application contexts to create applications that are more portable.

Chapter 4 *An Example Application*, describes a complete application, in several iterations. First, it shows a simple version of the program, a bitmap editor, as it would be written assuming the existence of a BitmapEdit widget (which is actually developed in Chapter 5). Then, two refined versions are developed, each demonstrating additional Toolkit programming techniques. Finally, the same application is shown as it would be written if the bitmap editor were implemented in an application window rather than with the BitmapEdit widget, as it would be written if no BitmapEdit widget existed.

The next two chapters describe widget internals, and the process of creating new widgets. While this information is not essential for all application programmers, many applications require a custom widget to implement their special graphics capabilities.

Chapter 5 *Inside a Widget*, describes the code inside a widget. Much of this code is common to all widgets. You can think of it as a framework that Xt uses to implement a widget's features. After reading this chapter, you should understand the procedure for creating your own widget around this framework.

Chapter 6 *Basic Widget Methods*, describes a widget's `initialize`, `expose`, `set_values`, `destroy`, `resize` and `query_geometry` methods. (A widget's methods are internal routines called automatically by Xt to give the widget a degree of independence from the application.) The chapter explains when Xt calls each method, and describes in detail what should be in each of these methods. Among other things, these methods prepare for and do the drawing of graphics that appear in a widget. This chapter describes what the Toolkit adds to the graphics model provided by Xlib but does not describe in detail how to draw using Xlib; this topic is described in Chapters 5, 6, and 7 of Volume One, *Xlib Programming Manual*.

Later chapters treat various topics of interest to either application or widget programmers, or both. Some of these topics have been introduced in the earlier chapters and are explored more completely in the later ones.

Chapter 7 *Events, Translations, and Accelerators*, describes the complete syntax of translation tables, which allow the user to configure the mapping of event sequences into widget actions. It also describes accelerators, a mechanism for mapping events in one widget to actions in another.

Chapter 8 *More Input Techniques*, describes how to handle events with event handlers and how to use information from the event structure inside an event handler or action routine. It also describes how to get file, pipe, or socket input, how to

use timeouts to call a function after a delay or at particular intervals, and how to use work procedures to do background processing. Finally, it discusses some low-level features of Xt for directly interacting with the event queue.

Chapter 9 *Resource Management and Type Conversion*, is a more thorough discussion of how resources work and how they should be used. This chapter describes in detail the resource file format and the rules that govern the precedence of resource settings. It also describes how to add your own type converter so that you can set application- or widget-specific data through resources. Finally, it describes subresources and how to use them.

Chapter 10 *Interclient Communications*, discusses communication through the X server between application and the window manager, and between two applications. The application-window manager communication is performed by code in the Shell widget. The application sets shell resources to control this communication with the window manager. Application-application communication is usually done with a process called selections. This form of communication is already implemented in most widgets that display text, but you may want to implement it in your own custom widgets. Selections can also pass other kinds of data such as graphics.

Chapter 11 *Geometry Management*, discusses how composite and constraint widgets manage the layout of widgets, and how to write your own simple composite and constraint widgets.

Chapter 12 *Menus, Gadgets, and Cascaded Pop Ups*, describes how menus work and describes several ways to create menu widgets. One of these ways involves the use of windowless widgets, or gadgets. This chapter also describes how to use more advanced features of the Xt pop-up mechanism, including modal cascades, to implement cascading pop-up menus.

Chapter 13 *Miscellaneous Toolkit Programming Techniques*, describes various Xt functions that have not been treated elsewhere in the book. These include functions for error and warning handling, case conversion, and so on.

Appendix A *OPEN LOOK and Motif*, provides an overview of the widgets available in AT&T's OPEN LOOK widget set and OSF's Motif. These widgets are contrasted with those in the Athena widget set.

Appendix B *Specifying Fonts and Colors*, gives information on the values that can be used when specifying fonts and colors as resources.

Appendix C *Naming Conventions*, describes a suggested set of conventions for naming widgets and elements with widget code.

Appendix D *Release Notes*, describes the changes between Release 3 and Release 4. This manual describes Release 4, but notes the most important changes since Release 3 for those familiar with that release.

Appendix E *The xbitmap Application*, shows the complete code for all versions of the *xbitmap* application and the BitmapEdit widget, which are described in Chapters 4 and 5.

Appendix F *Sources of Additional Information*, lists where to get the X software, lists companies that offer training in X programming, and describes additional books on the subject that have been or soon will be published.

Glossary gives you somewhere to turn should you run across an unfamiliar term. Some care has been taken to see that all terms are defined where they are first used in the text, but not everyone will read the manual in sequential order.

Master Index provides a thorough, combined index to Volumes Four and Five, making it easy to look up all the appropriate references to a topic, in either volume.

Volume Five consists of a permuted index, reference pages to each library function, and appendices that cover macros, structures, and defined symbols.

Font Conventions Used in This Manual

Italics are used for:

- UNIX pathnames, filenames, program names, user command names, and options for user commands
- New terms where they are defined

`Typewriter Font` is used for:

- Anything that would be typed verbatim into code, such as examples of source code and text on the screen
- The contents of include files, such as structure types, structure members, symbols (defined constants and bit flags), and macros
- Xt, Xaw, and Xlib functions
- Names of subroutines in the example programs

`Italic Typewriter Font` is used for:

- Arguments to functions, since they could be typed in code as shown but are arbitrary

Helvetica Italics are used for:

- Titles of examples, figures, and tables

Boldface is used for:

- Chapter and section headings

Requests for Comments

To help us provide you with the best documentation possible, please write to tell us about any flaws you find in this manual or how you think it could be improved.

Our U.S. mail address, e-mail address, and phone numbers are as follows:

O'Reilly & Associates, Inc.
632 Petaluma Avenue
Sebastopol, CA 95472
800-338-6887
international +1 707-829-0515

UUCP: uunet!ora!adrian Internet: adrian@ora.com

Bulk Sales Information

This manual is being resold by many workstation manufacturers as their official X Window System documentation. For information on volume discounts for bulk purchase, call O'Reilly and Associates, Inc., at 800-338-6887 or send e-mail to linda@ora.com (uunet!ora!linda).

For companies requiring extensive customization of the book, source licensing terms are also available.

Obtaining the X Window System Software

The X Window system is copyrighted but freely distributed. The only restriction this places on its use is that the copyright notice identifying the author and the terms of use must accompany all copies of the software or documentation. Thanks to this policy, the software is available for nominal cost from a variety of sources. See Appendix F, *Sources of Additional Information*, for a listing of these sources.

Example Programs

An early version of the example programs in this book is on the X11 Release 4 distribution in the contributed section. There are many ways of getting this distribution; most are described in Appendix F, *Sources of Additional Information*.

The current example programs are available free from UUNET (that is, free except for UUNET's usual connect-time charges). If you have access to UUNET, you can retrieve the source code using *uucp* or *ftp*. For *uucp*, find a machine with direct access to UUNET, and type the following command:

```
uucp uunet\!~/nutshell/xt/xtprogs2.tar.Z yourhost!~/yourname/
```

The backslashes can be omitted if you use the Bourne shell (*sh*) instead of *csh*. The file should appear some time later (up to a day or more) in the directory */usr/spool/uucppublic/**yourname***.

You don't need to subscribe to UUNET to be able to use their archives via UUCP. By calling 1-900-468-7727 and using the login "uucp" with no password, anyone may uucp any of UUNET's on line source collection. (You may wish to start by copying *uunet!/usr/spool/ftp/ls-lR.Z*, which is a compressed index of every file in the archives.) As of this writing, the cost is 40 cents per minute. The charges will appear on your next telephone bill.

You don't need to subscribe to UUNET to be able to access its archives by *ftp* either. However, you need to use a machine connected to the internet. To use *ftp*, *ftp* to *uunet.uu.net* and use *anonymous* as your user name and *guest* as your password. Then type the following:

```
cd nutshell/xt
binary (you must specify binary transfer for compressed files)
get xtprogs2.tar.Z
bye
```

The file is a compressed tar archive. To restore files once retrieving the archive, type:

```
uncompress xtprogs2.tar
tar xf xtprogs2.tar
```

The examples will be installed in subdirectories under the current directory, one for each chapter in the book. Imakefiles are included. (Imakefiles are used with *imake*, a program supplied with the X11 distribution that generates proper Makefiles on a wide variety of systems.)

All the application-defaults files are in the main examples directory. The application-defaults files are not automatically installed in the system application-defaults directory (usually */usr/lib/X11/app-defaults* on UNIX systems). (See Chapter 9, *Resource Management and Type Conversion*, for details.) If you have permission to write to that directory, you can copy them there yourself. Otherwise, you can set the XAPPLRESDIR environment variable to the complete path of the directory where you installed the examples. The value of XAPPLRESDIR must end with a / (slash). (Most of the examples will not function properly without the application-defaults files.)

Acknowledgments

As mentioned above, this manual includes some material from the *Xt Toolkit Intrinsics—C Language Interface*, by Joel McCormack, Paul Asente, and Ralph Swick. This is the document that defines the X Consortium standard for Xt, known as the Xt specification. Overt borrowings from the Xt specification are rare in this volume. However, the Xt specification document, as well as the sample code of Xt and the Athena widget set distributed with releases of X, provides the intellectual basis for most of what appears here.

We'd like to thank Sony Microsystems for the loan of a Sony NEWS workstation running their implementation of the X Window System. The speed and power of the Sony workstation, and the support of Sony's staff, were a great help in developing these books.

Additional development was done on a Sun-3 workstation running MIT's sample server, a Visual 640 X Display Station, and an NCD16 Network Display Station.

We would also like to thank the reviewers of the Alpha draft of this book, even though we almost had to start over because of their comments. They were David Lewis of Integrated Computer Solutions (ICS), Wendy Eisner of Sunquest Information Systems, Dan Heller of Island Graphics, Inc. (now working with O'Reilly and Associates), Miles O'Neal of Systems and Software Solutions, Inc., and Chris Peterson of MIT Project Athena (now of the X Consortium). Ian Darwin of SoftQuad and Bradley Ross of Cambridge Computer Associates reviewed the Beta draft. Extra thanks are due to Ralph Swick, Chris Peterson, and Robert Scheifler, who answered many questions during the development of this book.

Of course, we alone take responsibility for any errors or omissions that remain.

Special thanks go to Mark Langley, who wrote an early draft of this book. He helped to educate us about the Toolkit, and his efforts to make the book a success did not go unnoticed.

Of course, the authors would like to thank the entire staff of O'Reilly and Associates, Inc., and Cambridge Computer Associates, Inc., especially Ellie Cutler, Kathryn Ellis, Virginia Estabrook, Edie Freedman, Daniel Gilly, Kismet McDonough, Lenny Muellner, Linda Mui, Susan Sarkes Dalton, Ruth Terry, Sue Willing, and Donna Woonteiler, for their help and support in the development of this book. Dan Heller helped us learn how the Toolkit is really used in typical applications, and helped design the examples and the order of presentation. The illustrations (except where borrowed from other books in the series) were done by Andrea Pluhar and Chris Reilley. Tim O'Reilly developed the master index, and Dale Dougherty revised the indexing program to permit multi-volume indexing.

Perhaps most of all, we would like to thank our readers and customers for their patience, which we tested by promising that this book would be finished next month—every month for the last eight months. These days, with word processors, it is easy to generate a book-length manuscript, but no easier than it ever was to carve that text into something worth reading. We have had that book-length manuscript for a year, but have not been satisfied until now that it presented the material in a clear, friendly, and authoritative manner. We hope that the extra time we have spent boiling down the facts about this very new and continuously advancing subject will prove worthwhile to you, the reader.

—Adrian Nye and Tim O'Reilly

1

Introduction to the X Window System

This chapter introduces many of the most important concepts on which the X Window System is based, and describes the environment in which the X Toolkit operates. This chapter assumes that you are new to programming the X Window System. If you already have some experience programming the X Window System, you may wish to skim this Chapter for a brief review or even begin with Chapter 2.

In This Chapter:

The Server and Client 6
The Software Hierarchy 8
Event-driven Programming 10
The Window Manager 11
Extensions to X 12

1
Introduction to the X Window System

The X Window System (or simply X)* is a hardware-independent and operating-system-independent windowing system. It was developed jointly by MIT and Digital Equipment Corporation, and has been adopted by the computer industry as a standard for graphics applications.

X controls a "bit-mapped" display in which every pixel (dot on the screen) is individually controllable. This allows applications to draw pictures as well as text. Until recently, individual control of screen pixels was widely available only on personal computers (PCs) and high-priced technical workstations. Most general-purpose machines were limited to output on text-only terminals. X brings a consistent world of graphic output to both PCs and more powerful machines. Figure 1-1 compares an X application to an application running on a traditional text terminal.

Like other windowing systems, X divides the screen into multiple input and output areas called *windows*. Using a terminal emulator, windows can act as "virtual terminals," running ordinary text-based applications. However, as shown in Figure 1-1, windows can also run applications designed to take advantage of the graphic power of the bitmapped display.

X takes user input from a *pointer*. The pointer is usually a mouse but could just as well be a track-ball or a tablet. The pointer allows the user to control a program without using the keyboard, by pointing at objects drawn on the screen such as menus and command buttons. This method of using programs is often easier to learn than traditional keyboard control, because it is more intuitive. Figure 1-2 shows an application with a typical three-button pointer being used to select a menu item.

Of course, X also handles keyboard input. The pointer directs keyboard input from window to window. Only one window at a time can receive keyboard input.

In X, as in many other window systems, each application need not (and usually does not) consist of only a single window. Any part of an application can have its own separate subwindow, which simplifies the management of input and output within the application code. Such *child windows* are visible only within the confines of their parent window.

*The name "X Windows" is frowned upon by the developers of X.

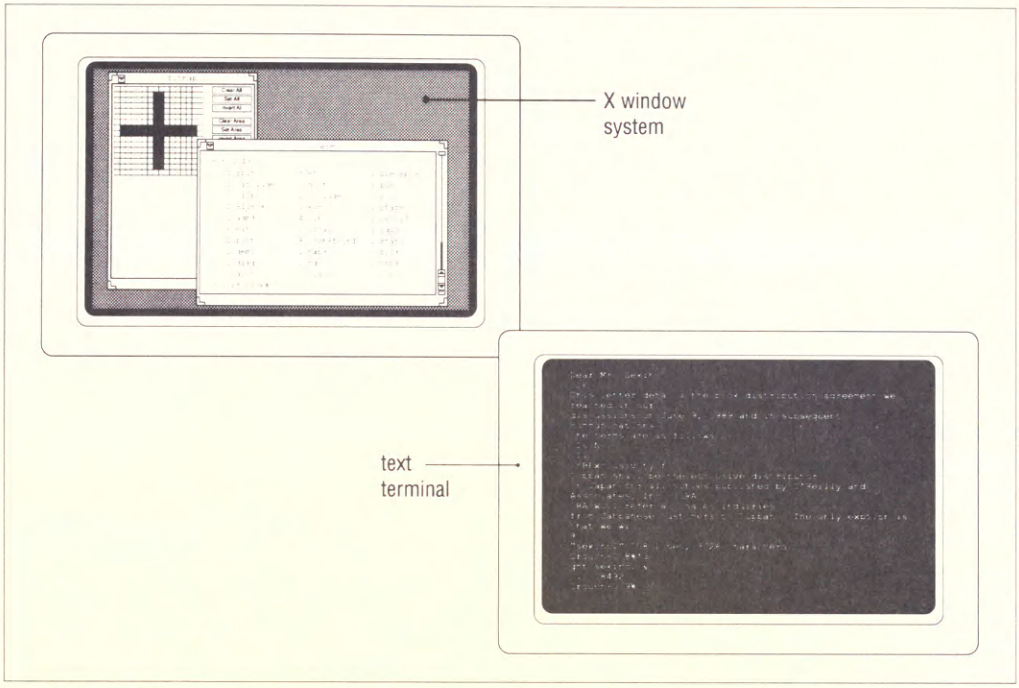


Figure 1-1. An X application, and an application on a traditional text terminal

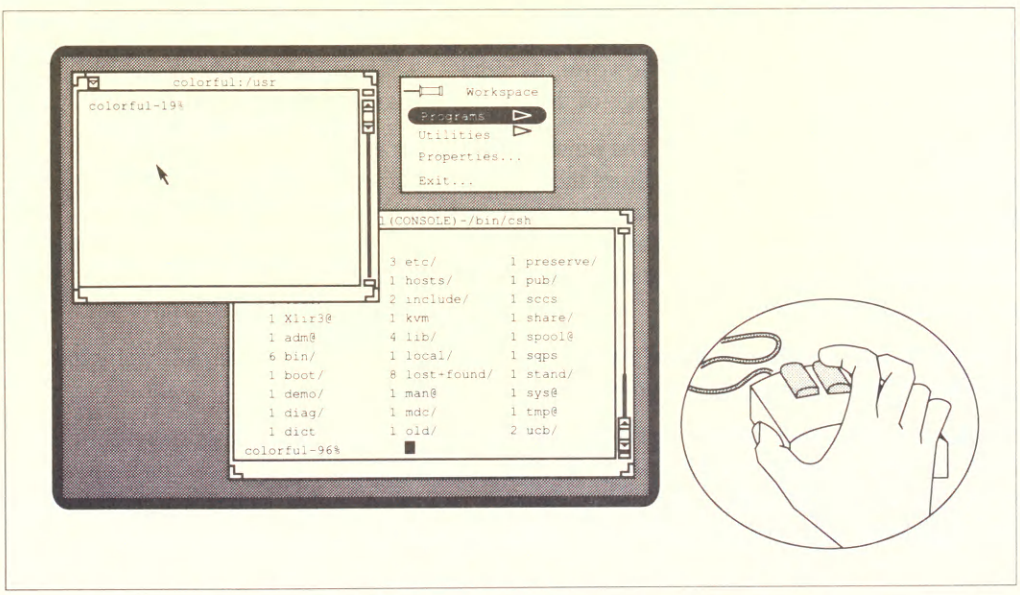


Figure 1-2. A three-button mouse directing the pointer to select a menu item

Windows are rectangular and oriented along the same axes as the edges of the display. Each window has its own coordinate system, with the origin in the upper-left corner of the window inside its border. The application or the user can change the dimensions of windows. Figure 1-3 shows a typical screen with several virtual terminals running. The screen also shows some applications, such as *xmh*, *xclock*, and *xload*, that run in their own windows.

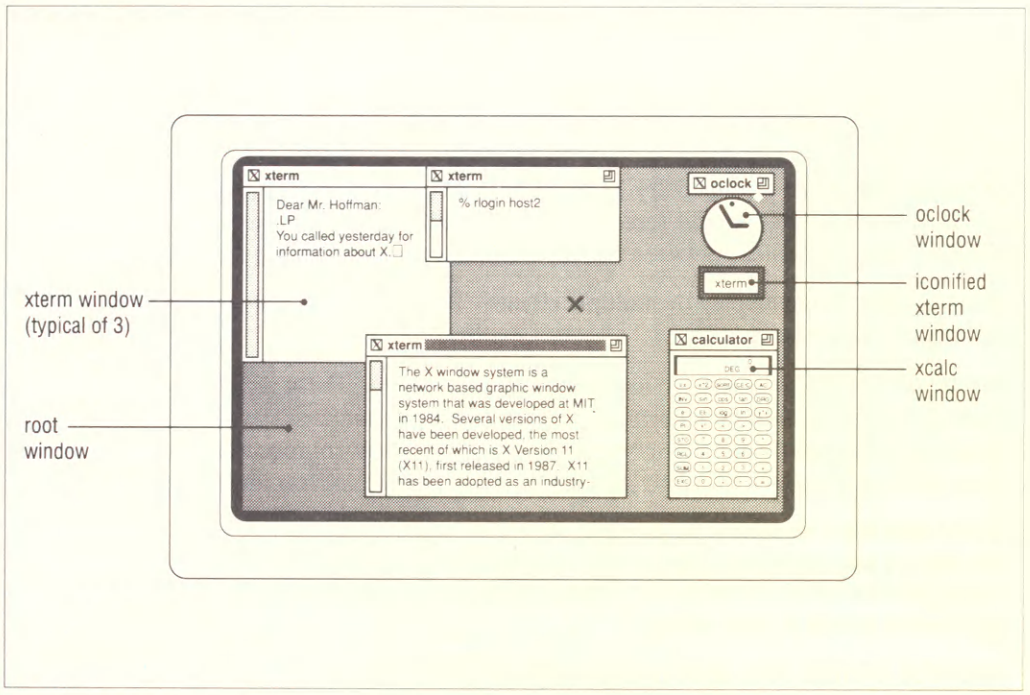


Figure 1-3. Screen layout of a typical user's X Window System

X supports both color and black-and-white displays.

Many of the above characteristics are also true of several other window systems. What is unusual about X is that it is based on a network protocol instead of on system-specific procedure and system calls. This network protocol enables X to be ported to different computer architectures and operating systems; it also allows programs to run on one architecture or operating system while displaying on another. Because of its unique design, X can make a network of different computers cooperate. For example, a computationally intensive application might run on a supercomputer, but take input from and display output on a workstation connected across a local area network. To the user, the application would simply appear to be running on the workstation.

1.1 The Server and Client

To allow programs to be run on one machine and display on another, X was designed as a network protocol—a predefined set of requests and replies—between two processes, one of which is an application program called a *client*, and the other of which, the *server*, controls the display hardware, keyboard, and pointer.

The user sits at the machine running the server. At first, this usage of the term server may seem a little odd, since file and print servers are normally remote machines. But the usage is consistent. The local display is accessible to other systems across the network, and for those systems the X server does act like other types of server.

The X server acts as an intermediary between user programs and the resources of the local system such as the keyboard and screen. It contains all device-specific code, and insulates applications from differences between display hardware. It performs the following tasks:

- Allows access to the display by multiple clients. The server may deny access from clients running on certain machines.
- Interprets network messages from clients and acts on them. These messages are known as requests. Some requests command the server to do two-dimensional drawing and move windows, while others ask the server for information. Protocol requests are generated by client calls to Xlib, either directly or through Xt and other function libraries.
- Passes user input to clients by sending network messages known as *events*, which represent key or button presses, pointer motion, and so forth. Events are generated asynchronously, and events from different devices may be intermingled. The server must pass the appropriate events to each client.
- Maintains complex data structures, including windows and fonts, so that the server can perform its tasks efficiently. Clients refer to these abstractions by ID numbers. Server-maintained abstractions reduce the amount of data that has to be maintained by each client and the amount of data that has to be transferred over the network.

In X, the term *display* is often used as a synonym for server, as is the combined term *display server*. However, the terms display and screen are not synonymous. A *screen* is the actual hardware on which the graphics are drawn. A server may control more than one screen. For example, a single server might control both a color screen and a monochrome screen, allowing a user to debug an application on both types of screens without leaving seats.

The user programs displaying on the screen(s) managed by a server are called its clients. There may be several clients connected to a single server. Clients may run on the same machine as the server if that machine supports multitasking, or clients may run on other machines in the network. In either case, the *X Protocol* is used by the client to send requests to draw graphics or to query the server for information, and is used by the server to send user input and replies to information requests back to the client.* All communication from the

*The X Protocol runs on top of any lower-level network protocol that provides bidirectional communication, and delivers bytes unduplicated and in sequence. TCP/IP and DECnet are the most common low-level network protocols currently supported by X servers.

client to the server and from the server to the client takes place using the X Protocol. The communication path between a client and the server is called a *connection*.

It is common for a user to have programs running on several different hosts in the network, all invoked from and displaying their windows on a single screen (see Figure 1-4). Clients running remotely can be started from the remote machine or from the local machine using the network utilities *rlogin* or *rsh*.

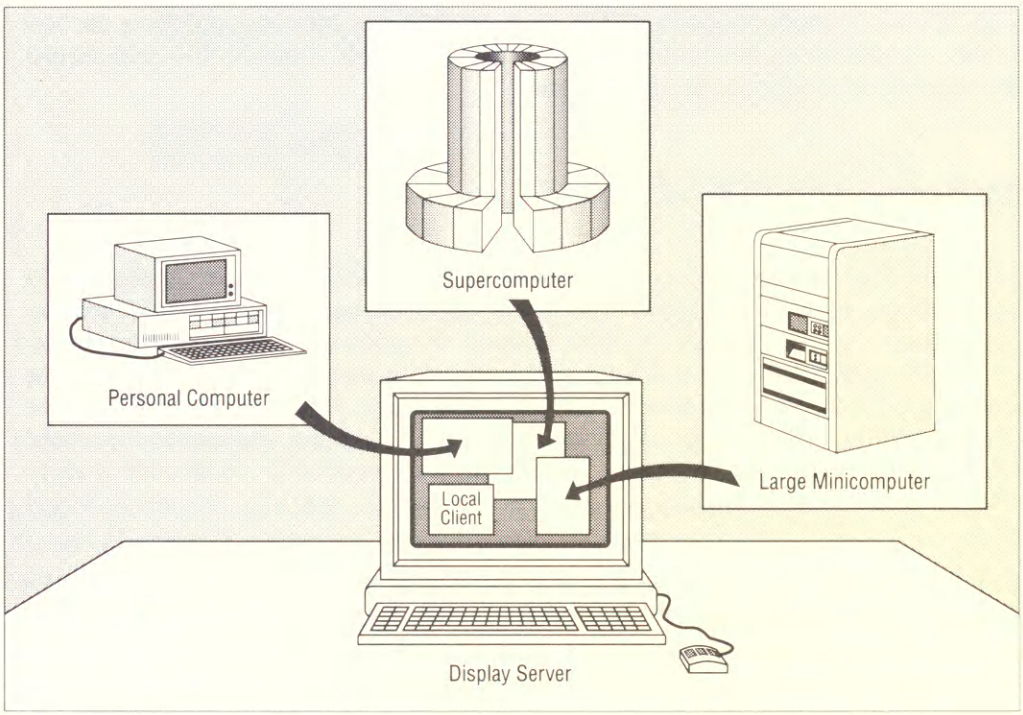


Figure 1-4. Applications can run on any system across the network

This use of the network is known as *distributed processing*. The most important application of this concept is to provide graphic output for powerful systems that don't have their own built-in graphics facilities. However, distributed processing can also help solve the problem of unbalanced system loads. When one host machine is overloaded, the users running clients on that machine can arrange for some of their clients to run on other hosts. Eventually, there may be automatic load-balancing applications, but currently, such remote execution is done manually. It is not unusual to see users in the X environment having several *xload* load monitor applications running on various systems throughout the network but displaying on their screen, so that they can see the balance of loads throughout the network.

Before leaving the subject of servers and clients, we should mention PC servers and X terminals. Software is available that allows various types of PCs to operate as X servers.* X terminals are special-purpose devices designed to run just an X server, and to connect to remote systems over a local area network. PC servers and X terminals are the least expensive way to provide an X screen for a user. Since most PCs use single-tasking operating systems, they can't run any clients at the same time as the server. Therefore, they too require a network adapter to connect to another system where clients are run.

X terminals and PC servers both demonstrate the strength of X's client-server model. Even though PCs and X terminals aren't able to do multitasking on their own, they give the user the effect of multitasking workstations, because they can interact simultaneously with several clients running on remote multitasking systems.

1.2 The Software Hierarchy

This book is about writing client applications for the X Window System, in C, using the Xt Intrinsics library and a set of widgets. This is only one of the many ways to write X applications, because X is not restricted to a single language or operating system. The only requirement of an X application is that it generate and receive X protocol messages according to the X Consortium Protocol specification.† However, using the Xt Intrinsics and a widget set is, and is expected to continue to be, the most common way of writing applications for several reasons: it is quite powerful; it results in applications that cooperate well with other X applications; it supports several popular user-interface conventions; and the C language is widely available.

Figure 1-5 shows the layering of software in an application that uses the Xt Intrinsics and a widget set. Notice that the Intrinsics are based upon Xlib, the lowest-level C-language interface to X. Xlib provides full access to the capabilities of the X protocol, but does little to make programming easier. It handles the interface between an application and the network, and includes some optimizations that encourage efficient network usage.

Xt is built upon Xlib. The purpose of Xt is to provide an object-oriented layer that supports the user-interface abstraction called a widget. A *widget* is a reusable, configurable piece of code that operates independently of the application except through prearranged interactions. A *widget set* is a collection of widgets that provide commonly used user-interface components tied together with a consistent appearance and user interface (also called *look and feel*). There are several different widget sets from various vendors that are designed to work with Xt. The use of widgets separates application code from user-interface code and provides ready-to-use user-interface components such as buttons and scrollbars. Xt, widgets, and widget sets are described in much more detail in Chapter 2, *Introduction to the X Toolkit*.

*Companies such as Graphics Software Systems, Interactive Systems, and Locus Computing offer server implementations for IBM-compatible PCs. White Pine Software offers an X server that runs under Multifinder on the Macintosh. An Amiga server is available from GfxBase/Boing. X terminals are available from Visual Technology, NCR, Network Computing Devices (NCD), Tektronix, Graphon Corp, and others. The number of X products on the market is growing rapidly.

†Volume Zero, *X Protocol Reference Manual*, provides a conceptual discussion of the X protocol and its detailed specification.

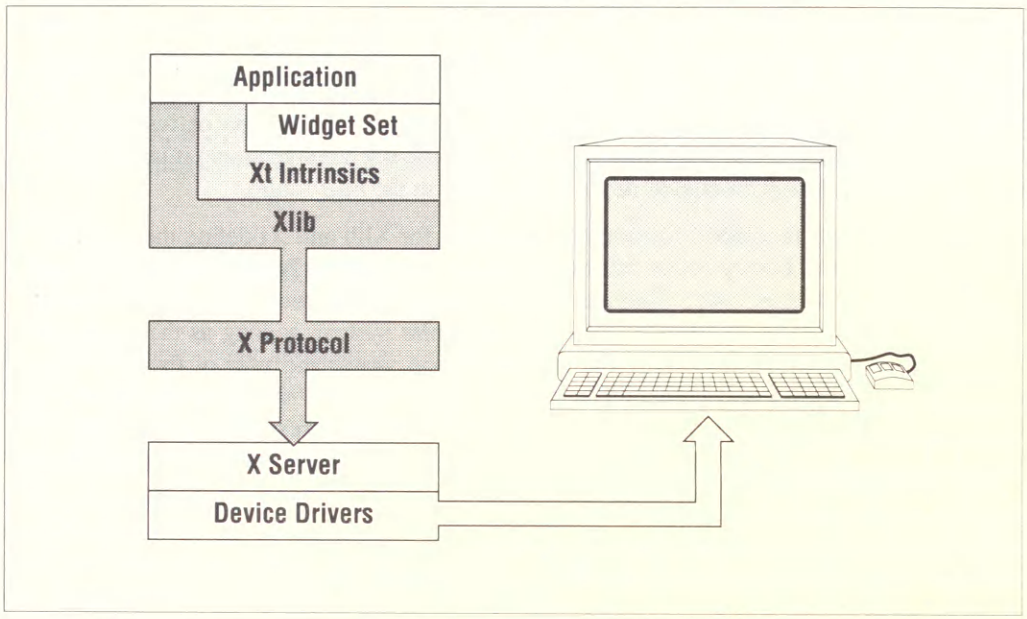


Figure 1-5. The software architecture of Xt Intrinsics-based applications

In this book, we'll refer to the combination of the Xt Intrinsics and one widget set as the *X Toolkit* or just the *Toolkit*. When referring to the Xt Intrinsics layer alone, we'll use *Xt*, or *the Intrinsics*.

Applications often need to call Xlib directly to accomplish certain tasks such as drawing. Xt does not provide its own graphics calls, nor does it provide access to every X protocol feature. This book will describe the features of Xlib that you may need from a Xt application, but it will not repeat the detailed description of Xlib programming found in Volume One, *Xlib Programming Manual*. You will find Volume One and Volume Two (*Xlib Reference Manual*) invaluable when you need to make Xlib calls.

Xlib, Xt, and several widget sets are available on MIT's public software distribution. The Motif and OPEN LOOK widget sets are not on the Release 3 or Release 4 distributions from MIT, but they are available for minimal cost from the vendors themselves (OSF and AT&T or Sun respectively.) The darkly shaded areas of Figure 1-5 indicate interfaces that are exclusive standards of the X Consortium. That Xlib is an exclusive standard means that computer manufacturers wishing to comply with the X Consortium standard must offer Xlib and cannot offer any other low-level X interface in C. The lightly shaded areas (such as the Xt Intrinsics) are nonexclusive standards—vendors are required to provide Xt but are also allowed to provide other toolkit-level layers for the C language. For example, Sun and AT&T will offer Xt, but will also offer XView as an alternate C-language toolkit-level layer. XView was originally designed for porting existing SunView™ applications to X, but it can also be used for writing new applications. Volume Seven, *XView Programming Guide*, describes programming with XView.

X software is unlike many other window systems in that it was designed to provide mechanism without mandating any certain style of user interface. In the words of its designers, X provides "mechanism without policy." The Xlib and Xt layers are standard because they can support any kind of interface. It is the widget set that actually imposes user-interface conventions, and it is this layer for which no standard has (yet) been considered by the X Consortium. However, because there is a strong need in the market for one or two standard widget sets that provide consistent appearance and user-interface conventions, it is likely that one or two widget sets will emerge as *de facto* standards in the near future.

It is important to note that the X Consortium standards for Xlib and Xt define the programming interface to each library (often referred to as the Application Programmer's Interface, or API), not the underlying code. That means that vendors are allowed to modify or rewrite the code to gain the best performance from their particular system, as long as they keep the programming interface the same. To you, the application writer and user of the Intrinsics, this means that you must always rely on documented behavior if you want your application to run on different systems. You must avoid accessing private structures, because they may be different in another vendor's release of the library, or they may be changed in a future release of X.

1.3 Event-driven Programming

Programming a graphically-based window system is fundamentally different from standard procedural programming. In traditional character-based interfaces, once the application starts, it is always in control. It knows just what kind of input it will allow, and may define exclusive modes to limit that input. For example, the application might ask the user for input with a menu, and use the reply to go down a level to a new menu, where the actions that were possible at the previous level are no longer available. Or a text editor may operate in one mode in which keyboard input is interpreted as editor commands, and another in which it is interpreted as data to be stored in an editor buffer. In any case, only keyboard input is expected.

In a window system, by contrast, multiple graphic applications may be running simultaneously. In addition to the keyboard, the user can use the pointer to select data, click on buttons or scrollbars, or change the keyboard focus from one application to another. Except in special cases (for example, where a "dialog box" will not relinquish control until the user provides some necessary information), applications are modeless—the user can suddenly switch from the keyboard to the mouse, or from one application area to another. Furthermore, as the user moves and resizes windows on the screen, application windows may be obscured or redisplayed. The application must be prepared to respond to any one of many different events at any time.

An X event is a data structure sent by the server that describes something that just happened that may be of interest to the application. There are two major categories of events: user input and window system side effects. For example, the user pressing a keyboard key or clicking a mouse button generates an event; a window being moved on the screen also

generates events—possibly in other applications as well if the movement changes the visible portions of their windows. It is the server's job to distribute events to the various windows on the screen.

Event-driven window programming reduces modes to a minimum, so that the user does not need to navigate a deep menu structure and can perform any action at any time. The user, not the application, is in control. The application simply performs some setup and then goes into a loop from which application functions may be invoked in any order as events arrive.

1.4 The Window Manager

Because multiple client applications can be running simultaneously, rules must exist for arbitrating conflicting demands for input. For example, does keyboard input automatically go to whichever window the pointer is in, or must the user explicitly select a window? How does the user move or resize windows?

Unlike most window systems, X itself makes no rules about this kind of thing. Instead, there is a special client called the *window manager* that manages the positions and sizes of the main windows of applications on a server's display. The window manager is just another client, but by convention it is given special responsibility to mediate competing demands for the physical resources of a display, including screen space, color resources, and the keyboard. The window manager allows the user to move windows around on the screen, resize them, and usually start new applications. The window manager also defines much of the visible behavior of the window system, such as whether windows are allowed to overlap or are tiled (side by side), and whether the keyboard keyboard focus simply follows the pointer from window to window, or whether the user must click a pointer button in a window to change the keyboard focus.

Applications are required to give the window manager certain information to help it mediate competing demands for screen space or other resources. For example, an application specifies its preferred size and size increments. These are known as *window manager hints* because the window manager is not required to honor them. The Toolkit provides an easy way for applications to set window manager hints.

The conventions for interaction with the window manager and with other clients have been standardized by the X Consortium as of July 1989 in a manual called the *Inter-Client Communication Conventions Manual* (ICCCM for short). The ICCCM defines basic policy intentionally omitted from X itself, such as the rules for transferring data between applications (selections), for transferring keyboard focus, for installing colormaps, and so on.

As long as applications and window managers follow the conventions set out in the ICCCM, applications created with different toolkits will be able to coexist and work together on the same server. Toolkit applications should be immune to the effects of changes from earlier conventions because the conventions are implemented by code hidden in a standard widget called Shell. However, you should be aware that some older applications and window managers do not play by the current rules.

1.5 Extensions to X

X is also *extensible*. The code includes a defined mechanism for incorporating extensions, so that vendors aren't forced to modify the existing system in incompatible ways when adding features. An extension requires an additional piece of software on the server side and an additional library at the same level as Xlib on the client side. After an initial query to see whether the server portion of the extension software is installed, these extensions are used just as Xlib routines and perform at the same level.

Among the extensions currently being developed are support for 2-D spline curves, for 3-D graphics, and for display PostScript™. These extensions can be used in Toolkit applications just as Xlib can.

2

Introduction to the X Toolkit

This chapter provides a conceptual introduction to the X Toolkit, followed by a practical tutorial that starts with the most fundamental toolkit program, a "hello world" type application consisting of only a single widget. This application is successively refined until the major elements of any X Toolkit program have been introduced.

In This Chapter:

Programming with Widgets 15
- Contents of a Widget Set 17
- Widget Classes and Instances 18
- Widget Configurability with Resources 23
- Widget Independence 25
- Widget-Application Interaction 27
- Xt and Object-oriented Programming (OOP) 30
  - The Object 30
  - Methods 31
  - Messages 31
  - Encapsulation 31

Structure of X Toolkit Applications 32
A Simple X Toolkit Application 33
- The Code 33
- Compiling the Application 36
- The Application-defaults File 38
- To Hardcode or Not to Hardcode 40

Connecting Widgets to Application Code 41
- Callbacks 42
- Actions 45
  - The Actions Table 48
  - Format of an Action Procedure 48

The Translation Table 49
Hardcoding Translations 51
More About Resources 52
Setting and Getting Resources from the Application 52
Setting Resources with the Varargs Interfaces 53
Setting Resources with the ArgList Interfaces 54
Getting a Resource Value 56
Core Resources 57
Other Inherited Resources 59

2
Introduction to the X Toolkit

Building applications for a graphical user interface using a low-level programming library such as Xlib requires much sophisticated programming. To simplify development, each of the user-interface elements of a graphical application—scrollbars, command buttons, dialog boxes, pop-up or pull-down menus (everything but the main application window) should ideally be available ready-made, so the programmer need only integrate them with the application code.

The purpose of the X Toolkit is to provide such a simplified approach to graphical user-interface programming. However, in keeping with the X philosophy of "mechanism, not policy," the designers of the X Toolkit didn't develop a fixed set of components with a predefined look and feel. Instead, they created a general mechanism for producing reusable user-interface components, which can be either built by the application programmer himself, or constructed by lower-level "object programmers" and collected in libraries for application use.

The heart of the Toolkit is a C library called Xt, also known as the X Toolkit Intrinsics. Xt provides routines for both creating and using user-interface components called widgets. A typical application will use Xt along with a library of pre-built widgets (a widget set) such as menus, dialog boxes, scrollbars, command buttons, and so forth, working together to provide a consistent application "look and feel."

2.1 Programming with Widgets

The simplest way to understand the role of widgets in an application is to look at a hypothetical application, such as the one shown in Figure 2-1.

If you are at all familiar with graphical applications, you will recognize many of the widgets called out in the figure as standard user-interface elements:

- *Command buttons*, which initiate some application action when clicked on with the mouse.
- A *scrollbar*, which allows the user to scroll the data visible in an associated display window. (The position of a "thumb" within the scrollbar indicates the position of the visible data within a larger data buffer, and lets the user change the current position in the buffer

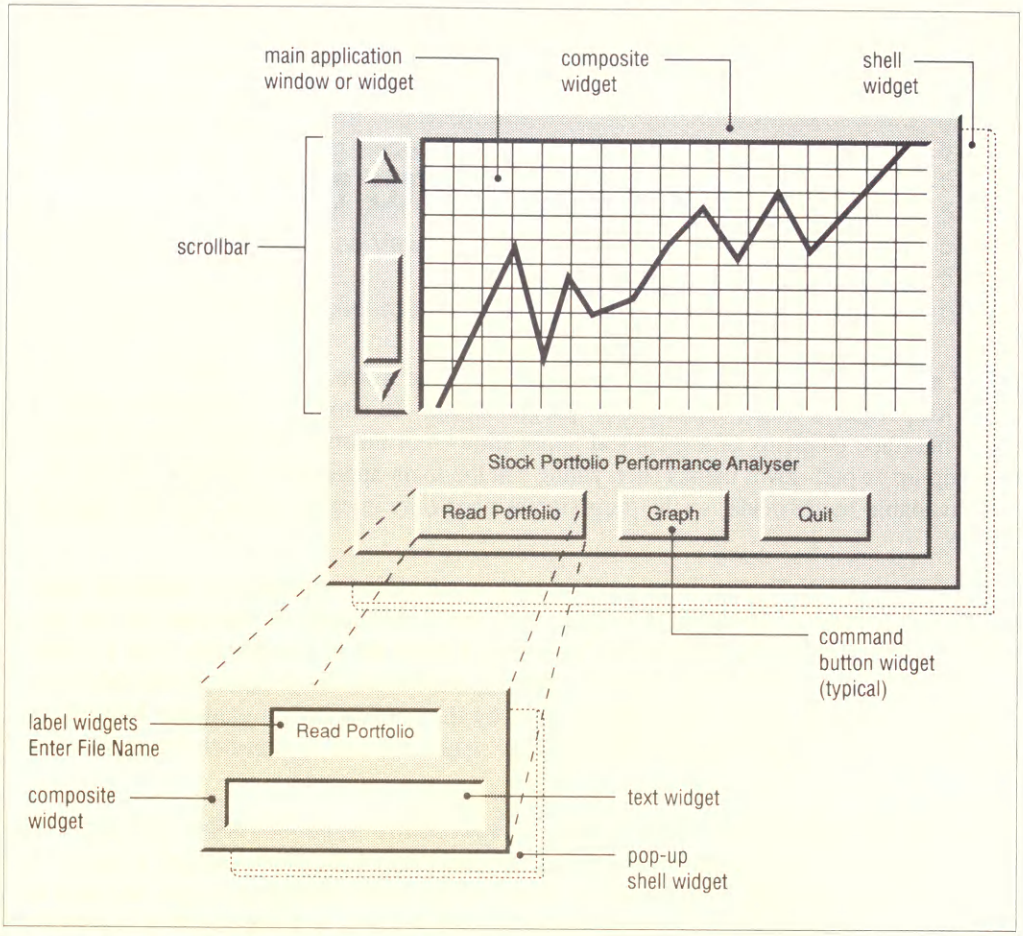


Figure 2-1. A widget-based application

by dragging the thumb with the mouse. Widget sets vary in their term for the scrollbar thumb.)

- A *data entry area*, in which the user can type information requested by the application.

There are other widgets in the application that might not be as obvious:

- *Composite widgets*, which are used to contain other widgets. In all widget sets, there are special classes of widgets whose function is to manage the position (and possibly the size) of the *child* widgets they contain.

  Composite widgets are an important part of any widget set, since they insulate the application programmer from having to place each widget individually, or from having to reposition or resize various widgets when the application is resized by the user. Composite widgets automatically adjust the layout of their children when child widgets are added or removed.

- *Shell widgets.* In any X Toolkit application, a special widget called a Shell widget is created by the call to initialize the Toolkit. This widget is used as the parent of all other application widgets (with the exception of pop ups, which receive their own transient Shell widget as a parent), and includes special functionality that allows it to interact with the window manager. The Shell widget is invisible, since it is overlaid by the main widget of the application (typically a composite widget), which is exactly the same size.*

- *A special-purpose application window.* Most applications have at least one window that has unusual characteristics not supported by an existing widget. For example, in our hypothetical application, the main application window is used to graph the performance of a stock portfolio.

  There are two ways to implement such windows. You can add functionality to an existing widget (usually what is called a Core widget) by adding actions as described and demonstrated in Chapter 4, *An Example Application*, or you can write your own widget, as described beginning in Chapter 5, *Inside a Widget*. As you will see, after you have written the application code to operate the special-purpose window, making a widget that does the same thing is mostly a matter of rearranging the code and changing variable names. Placing code into a widget gives you easier access to the configurability features of Xt, and neatly packages up the code to operate the special window.

- *A pop-up dialog box.* A *pop up* is a widget that appears temporarily on the screen, until the user provides a certain kind of input. Pop ups are usually invisible when the application starts up. Using Xt, applications create pop ups very much like permanent widgets. The only difference is that a special kind of Shell widget called a *pop-up shell* needs to be created as the parent of the widget to be popped up. Many types of menus and dialog boxes (also referred to as "notices") are intended to be used as pop ups.

2.1.1 Contents of a Widget Set

Shell widgets, and a few other base widget classes (Core, Composite, and Constraint) that are used to build more complex widgets, are defined by Xt. The special-purpose application window is written by the application programmer. All of the other widgets shown in Figure 2-1 are available in any widget set, though they have various names.

The Xaw library in the standard X distribution contains the Athena widget set. This book provides examples using the Athena widgets because they are universally available, and because the techniques for using, modifying, and creating widgets are the same, regardless of which widget set you use. However, the Athena widget set was not intended to be complete—it was built mainly for testing and demonstrating the Intrinsics. Furthermore, the Athena set does not have a particularly attractive appearance. For these reasons, we suggest that serious application development efforts should begin with a commercial widget set.

*The Shell widget is actually a type of composite widget, which is designed to have only one child widget. Its layout policy is extremely simple. It just makes its child widget exactly the same size as itself.

Two commercial widget sets that are easily available and quite complete are OSF's Motif and AT&T's OPEN LOOK widgets. Both Motif and OPEN LOOK contain menus, scrollbars, command buttons, dialog boxes, and a wide variety of composite widgets. Both have an attractive appearance and consistent, well defined user-interface conventions. Each comes with a style guide that contains suggestions for designing applications to blend in well with other applications using that widget set.

Appendix A, *OPEN LOOK and Motif*, describes these widget sets.

2.1.2 Widget Classes and Instances

A widget set defines *classes* of widgets. Both the Athena and Motif widgets sets have a Label widget class which performs approximately the same job. However, these are two different widget classes in Xt, even though they may seem to fit the standard English definition of a single class. (In practice, one would rarely want to use both of these widget classes in one application anyway.)

Command is a class of widget, as are Box, Label, Scrollbar, and the other Athena widgets shown in our hypothetical application. Each time you create a widget, you create an *instance* of one of these predefined classes. For example, you might create several Command widgets, each with a unique name, containing a unique text label, and each invoking different application code when it is clicked on. All these widgets would be of class Command. They would have similar characteristics, but they would not necessarily look or act exactly the same since each could have been configured differently.

A widget class has certain fixed features which are common to all instances of that class, and certain characteristics which can be changed from instance to instance.

How you view a class depends on whether you are using existing widget classes or writing new ones. Eventually you will thoroughly understand both these views of a class, since you will be a competent widget user and widget writer. Both views are introduced here because we don't want to mislead you by telling you only half the story of what a class is. But for the first four chapters of this book, we will be concentrating on the widget user's point of view.

For a user of existing widget classes (a widget set), a widget class is a black box that has certain fixed features and certain configurable features, both of which are documented on the widget class's reference page. You need not know anything about how a class is implemented in Xt. You know that when you create an instance of that widget class, the instance will have the documented fixed features and you can set the configurable features. A user of existing widget classes is most interested in the configurable features, since setting these is a big part of programming an Xt application. Each configurable feature is called a *resource*. Resources are more fully introduced in the next section.

If you are writing a widget class, or you have written one and see things from that perspective, a class seems slightly different. To you, a widget class is a set of three files in which the widget class is implemented. The widget class is no longer a black box—it is an open box. Knowing what widget class code looks like, you know that its fixed features and its configurable features are implemented in distinct sections of the code. You know that each resource is actually represented by a field in a structure. Even though you may not even have access

to the source code for a widget class, your definition of a class is based on how that class is implemented.

These two views of a class take on special relevance when looking at a characteristic of the Toolkit called class inheritance. Widget features and characteristics can be *inherited* from other, more basic classes of widgets. To a widget user, class inheritance is important only because it means that the resources (configurable features) of a widget class are defined not only by that class but also by the classes that class inherits features from, called its *superclasses*. When you look up a widget classes's features on its reference page, you will have to look up its superclass and look on that page also, and continue up the class inheritance hierarchy to the most basic widget, in order to get a complete list of its capabilities. This sounds difficult, but in reality you get to know the features of the most basic classes by heart, and there are not very many classes.

From the widget writer's point of view, inheritance means that a new class of widget has to define only its own unique features, and need not re-implement those common to all widgets, or already implemented by an existing superclass. All classes exist in a hierarchy, which defines from which other classes each class inherits features. (Note that the class hierarchy is completely different from the parent-child relationship of widget instances you create in an application. The class hierarchy of a particular widget set is fixed by the widget set writers, while the instance hierarchy is different in every application and is determined by the application writer.)

Figure 2-2 shows the class inheritance hierarchy for the Athena Widgets. Classes defined by Xt (which are the same for all widget sets) are shaded gray.*

The Core widget class, defined by the Intrinsics, is the root of the hierarchy, from which all other classes are descended. The Core class defines characteristics common to all widgets, such as size and position.

The Athena Simple widget class inherits basic widget features from Core and adds a few minor features of its own (for example, the ability to change the cursor shape when it enters the window). The Label widget in turn adds the ability to print a string, and mechanisms for changing the font and placement of the string. widget)" Command then inherits features from Label (including those already inherited from Core and Simple) and adds more features (the ability to accept user input, and to highlight the button). Command is known as a *subclass* of Label, and Label is the *superclass* of Command. In general, lower classes in the hierarchy have more features.

*Note that the X Toolkit supports only *single inheritance*. That is, characteristics can be inherited directly from only one superclass (though that superclass may itself have inherited characteristics from prior superclasses.

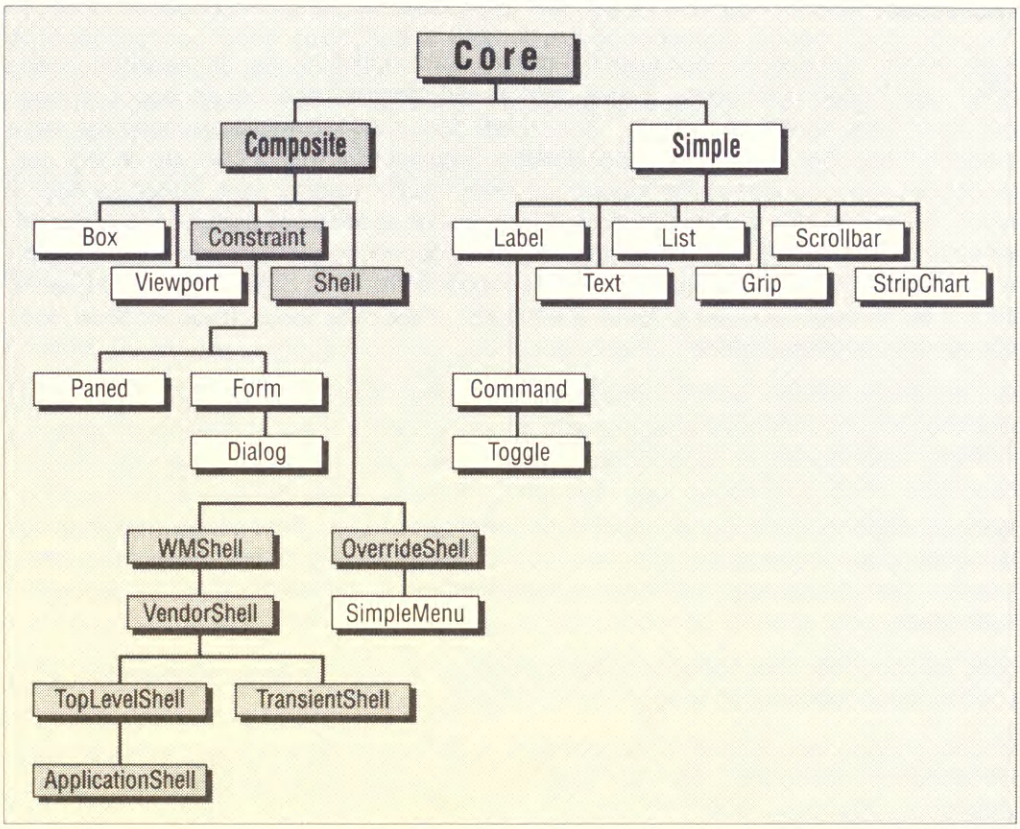


Figure 2-2. Inheritance among the Athena widgets

As you can see from Figure 2-2, Text, List, Scrollbar, and Grip, like Label, are simple widgets subclassed directly from Core. Command is a refinement of Label and Toggle is a refinement of Command. The Text widget provides a ready-built, configurable text editor. List displays a list of strings, which can be selected individually, and passed to an application function. (For example, this widget might be used to implement a file selection box from which the user might select which file to open.)

Scrollbar widgets are designed to be attached to other windows and to provide a mechanism by which the user can scroll the data in a window by dragging or clicking in the scrollbar. A Grip is a small solid box used for resizing panes in a Paned widget (see Figure 2-3).

The Paned widget manages a series of vertical panes, which have the constraint that they can only be resized in the vertical direction by using the pointer to drag Grip widgets provided for that purpose. Paned is useful for applications such as *xmh*, the X mail handler, that display data in several different windows simultaneously.

Figure 2-3 shows a Paned widget.

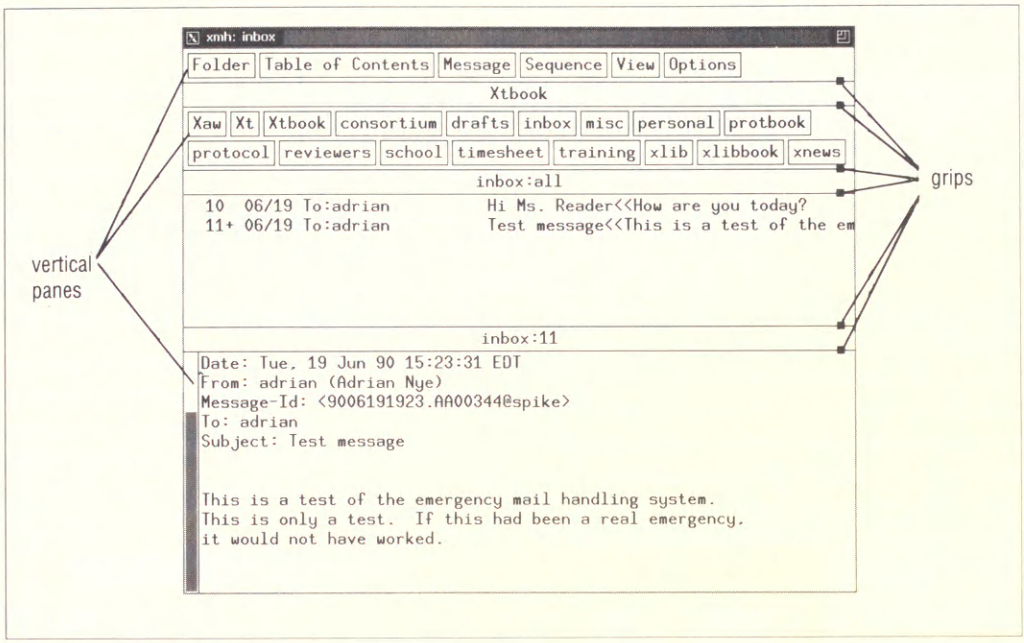


Figure 2-3. An Athena Paned widget

Form allows the position of its children to be specified relative either to each other or to fixed points in the Form parent. Children are thus constrained always to have the same relative position in the parent, even when it is resized.

Dialog is a subclass of Form that has been given specific children: a Label, which is used for printing a message, an optional Text widget for data entry, and two command widgets, for confirming or cancelling the action suggested by the dialog.

Figure 2-4 shows a Dialog widget. The Composite class adds geometry-management capabilities to the basic characteristics defined by Core. Box is a subclass of Composite that is mainly intended to manage a group of similarly-sized Command widgets. Viewport is a composite widget that provides a main window and horizontal and vertical scrollbars. Figure 2-5 shows a Viewport widget managing a List widget and a Scrollbar widget.

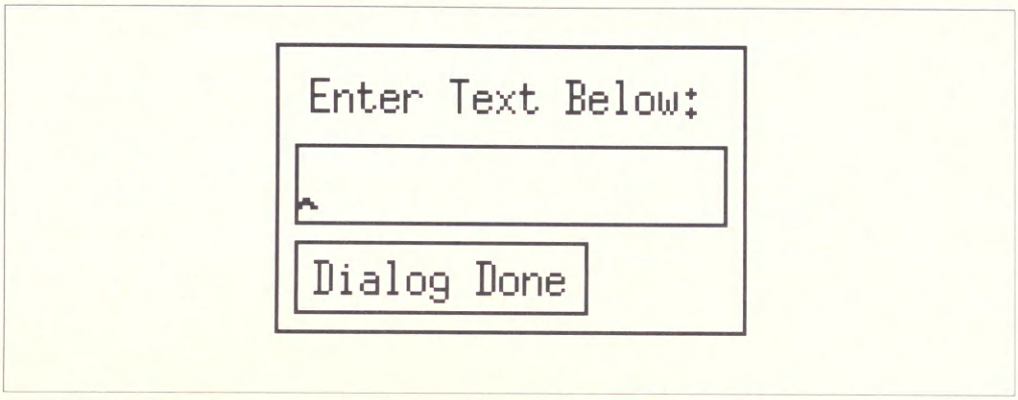


Figure 2-4. An Athena Dialog widget

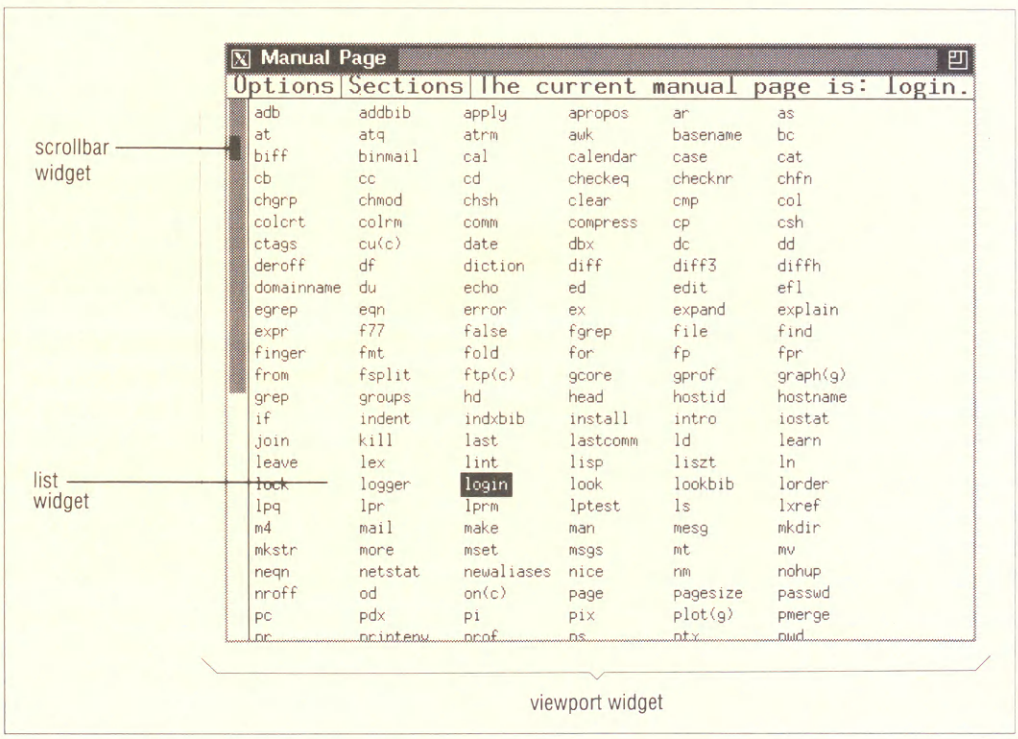


Figure 2-5. A Viewport widget

Shell is a special class of Composite widget designed for interaction with the window manager.

Most applications use a TopLevel shell for their main window; pop ups use either an OverrideShell or a TransientShell. (See Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*, for more information on Shell widget classes.)

Constraint is a further refinement of Composite that allows the application or the user to supply instructions about how the size and position of each child should be managed.

The Athena widgets are described in Volume Five, *X Toolkit Intrinsics Reference Manual*.

New widgets can be subclassed by the widget programmer directly from Core, Composite, or Constraint, or can be subclassed from an existing widget in any widget set that has some of the desired behavior. For example, it is easy to imagine creating subclasses of Form to provide other types of preconfigured dialog boxes or data entry screens.

As long as appropriate widget classes are available, the application programmer needs to know little or nothing about widget internals. In fact, even if widget internals are known, it is unwise to depend on them. Widgets should be treated as black boxes with documented inputs and outputs. If only these documented interfaces are used, the widget internals can be modified without affecting the application, and the application can be modified without affecting the widget.

.1.3 Widget Configurability with Resources

To serve their purpose as reusable user-interface components, widgets must be highly configurable. For example, an application programmer must be able to define not only a separate label for each Command widget, but also the application function that is invoked when the button is clicked. The programmer will also want to let the user define additional attributes such as font and color.

To support this degree of configurability, widget classes can declare variables as named *resources* of the widget. The application can pass the value of widget resources as arguments to the call to create a widget instance, or can set them after creation using the Intrinsics call `XtSetValues`. In addition, though, as an application starts up, a part of Xlib called the *resource manager* reads configuration settings placed in a series of ASCII files by the user and/or the application developer, and Xt automatically uses this information to configure the widgets in the application. The collection of resource name-value pairs contained in the various resource files and set directly by the application is collectively referred to as the *resource database*.

Figure 2-6 shows several Label widgets configured with different resource settings, to show you how radically the appearance of even such a simple widget can be altered. Note that a widget's input characteristics can also be configured (although the Label class has no input characteristics to configure).

The resource manager provides a very flexible mechanism for generalizing the behavior of widgets. The application developer can "hardcode" the value of resources that cannot be changed without crippling the widget or application, and can establish reasonable defaults for the rest, so that the user can configure all nonessential aspects of application appearance and behavior.

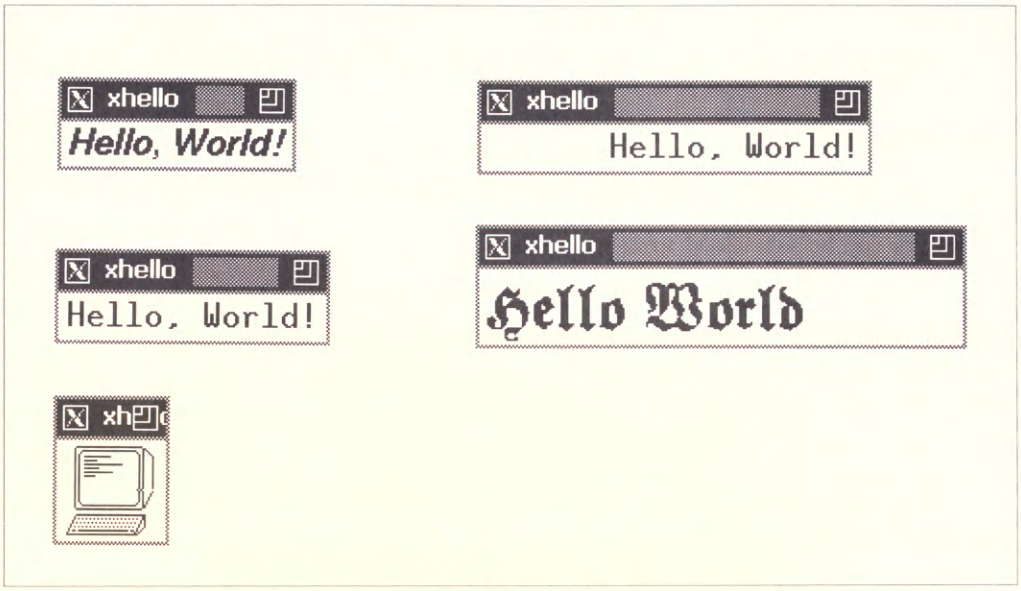


Figure 2-6. Several Label widgets configured using resources

Note that the term resource is used somewhat ambiguously in X. First, in the original documentation for Xlib and the X protocol, various data structures that are maintained by the server and identified to clients only by an integer ID are referred to as resources. These data structures include windows, bitmaps, fonts, and so forth.

Second, the term is commonly used to refer both to a widget variable publicly declared as a widget resource, and to the name-value pairs in the resource database. In this book, we will use the term *resource* to refer to the actual variable and its current value in a widget instance, and the term *resource setting* to refer to a name-value pair in the database. The two are closely related, but may not be identical. For example, separate settings for the same resource may be requested in an application-defaults file and in an individual user-preference file.* Furthermore, the value of a resource may be set on the fly by a call to `XtSetValues`, but this value is never saved in the resource database. (This value can be retrieved from within a widget or application by a call to `XtGetValues`.)

*There are several possible sources of resource settings. Which will actually take effect is determined by rules of precedence that are described (along with the various sources of resource settings) in Chapter 9.

2.1.4 Widget Independence

Each widget operates, to a large degree, independently of the application. Xt dispatches events to a widget, which performs the appropriate actions according to the design of its class, without application help. For example, widgets redraw themselves automatically when they become exposed after being covered by another window.* Widgets also handle the consequences when the values of their resources are changed. An instance of Label, for example, does not depend on the application that created it to determine its size. By default, Label will choose a size large enough to accommodate the current string in the current font. If the application changes the text or font in the Label widget with a call to `XtSetValues`, the Label widget itself will attempt to keep its own window large enough to accommodate the current string. (Of course, if necessary, the application can also explicitly choose the Label widget's size.) When the application tells a Label widget what font to display its string in, the widget knows how to load a new font, recalculate its own size, and redraw its string—the application doesn't have to micro-manage any of this. The application simply sets the font resource of the widget, and the widget does the rest.

Figure 2-7 and Figure 2-8 illustrate how a widget operates independently of the application and that `XtSetValues` lets the application set how a widget operates itself.

*Redrawing is necessary because only the visible contents of X windows are maintained by the X server. When one window is obscured by another, the contents of the obscured window are lost, and must be redrawn when it is exposed. X clients are responsible for redrawing the contents of their windows when this happens. Fortunately, Xt automatically redraws correctly written widgets at the appropriate times so that your application doesn't have to worry about it.

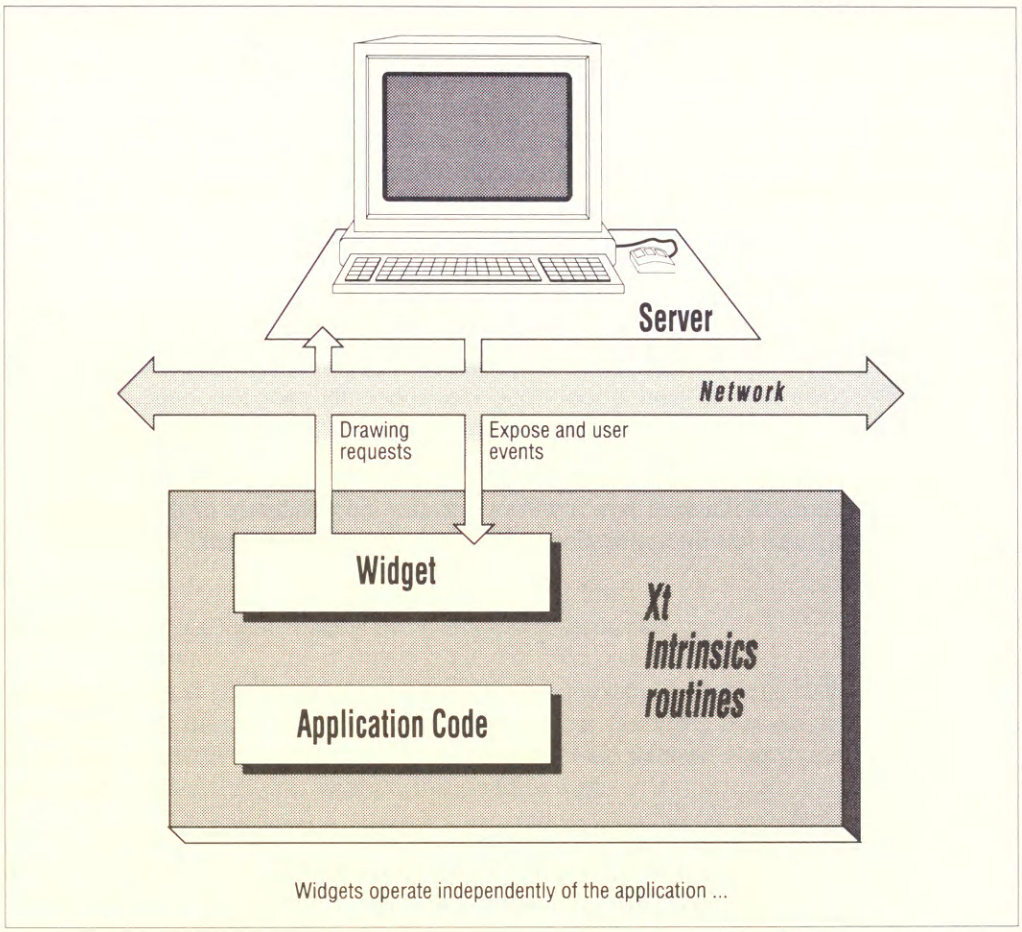


Figure 2-7. Widgets operate independently of the application

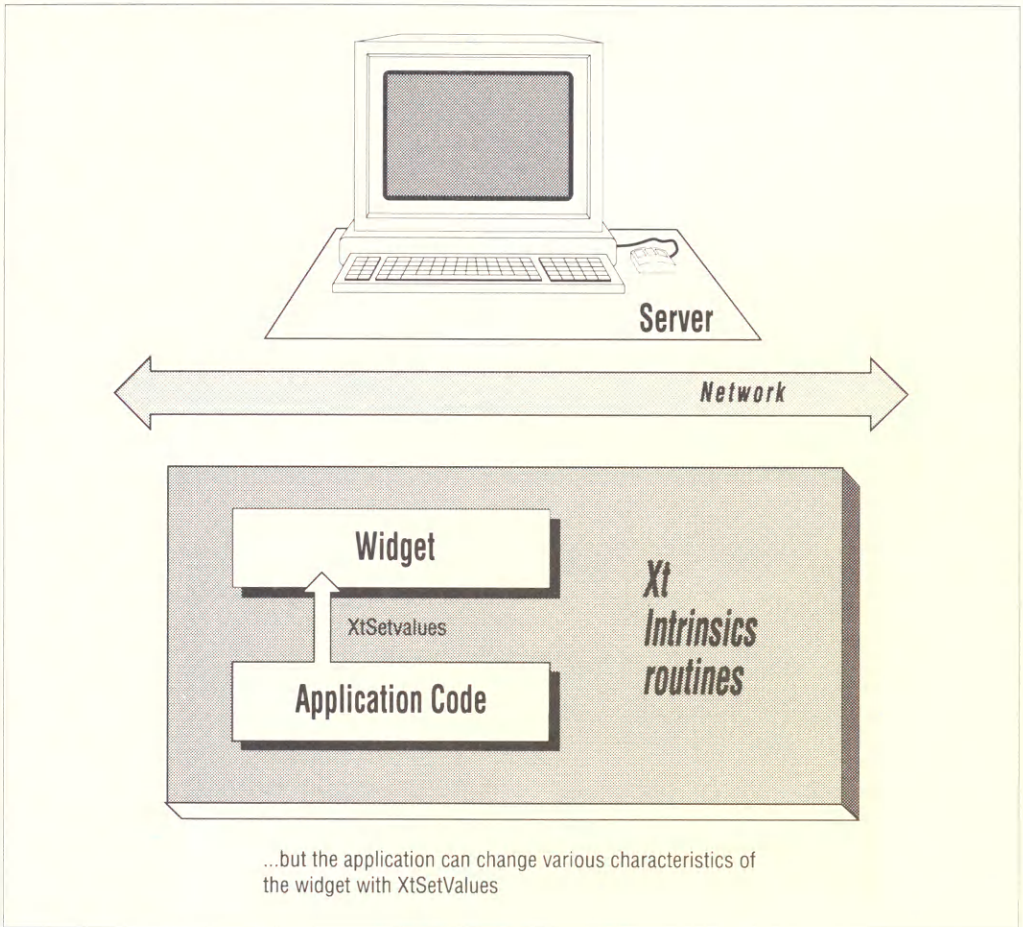


Figure 2-8. XtSetValues lets the application set how a widget will operate itself

2.1.5 Widget-Application Interaction

In the other direction, widgets are designed to let the user control the application. Therefore, widgets have the ability to invoke certain sections of application code—elements of the application's own choosing. Again, though, widgets will operate fine without invoking any application code, but if they don't invoke any, they won't do anything for the user.

One way that the application arranges for widgets to invoke application code is by registering application functions with Xt. Once the application is running, Xt will call these functions in response to some occurrence in the widget. For example, a Command widget usually invokes an application function when the user clicks on the widget. (Thus, the widget labeled "Quit" might invoke the code that exits the application.) Or the Scrollbar widget notifies the application when the user has moved the thumb, by calling a function that the application has registered with the widget for that purpose. Figure 2-9 and Figure 2-10 illus-

trate how an application registers a function during the startup phase, and how Xt then calls the function during the event-loop phase in response to a particular occurrence in the widget.

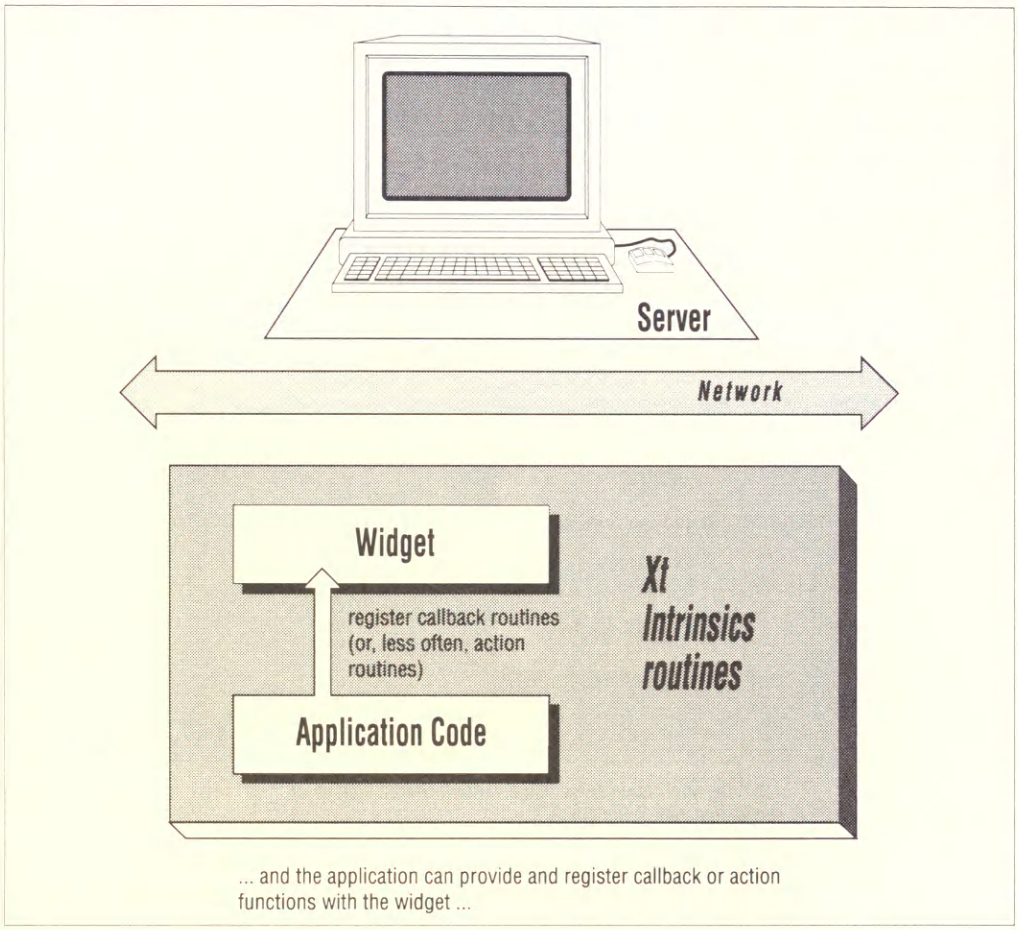


Figure 2-9. Application registers a function with Xt during startup phase

There are three separate mechanisms that can be used to link widgets and application functions: callbacks, actions, and event handlers.

Generally speaking, a widget expecting to interact with an application will declare one or more *callback lists* as resources; the application adds functions to these callback lists, which will be invoked whenever the predefined callback conditions are met. Callback lists are resources, so that the application can set or change the function that will be invoked.

Callbacks are not necessarily invoked in response to any event; a widget can call the specified routines at any arbitrary point in its code, whenever it wants to provide a "hook" for application interaction. For example, all widgets provide a `destroyCallback` resource to allow applications to interpose a routine to be executed when the widget is destroyed.

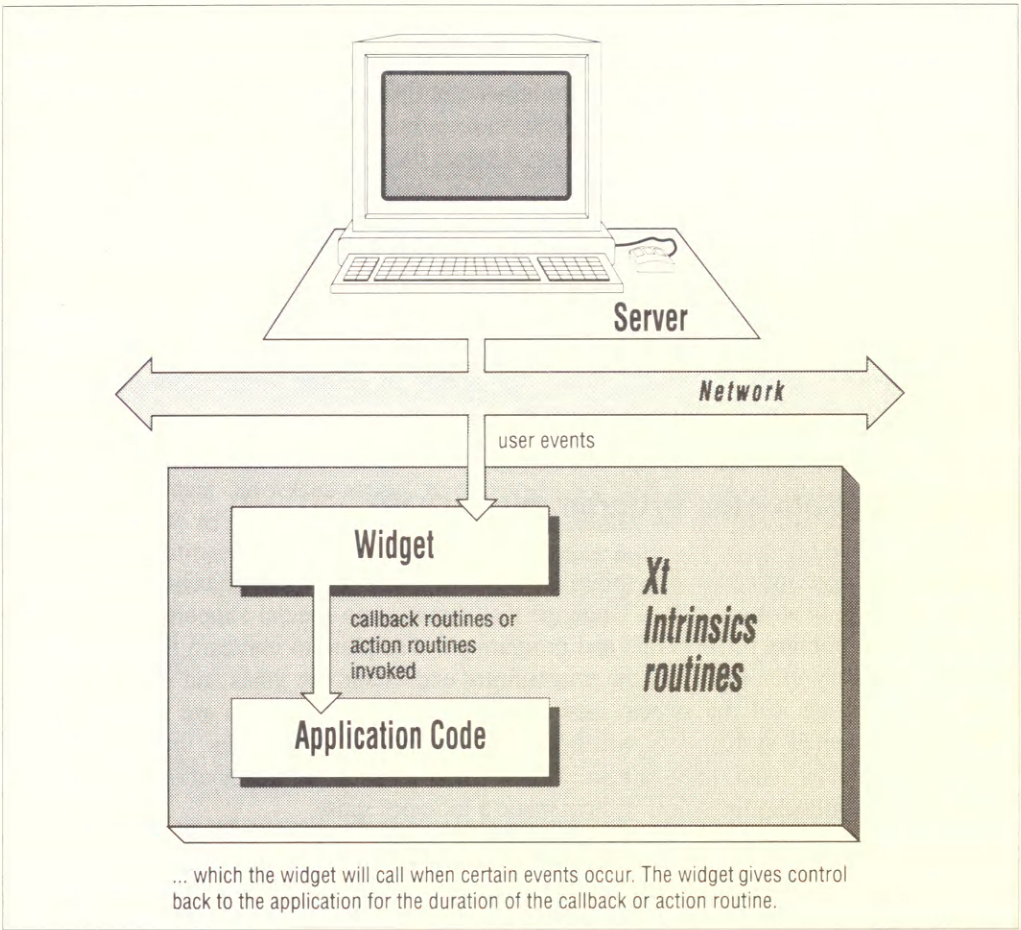


Figure 2-10. Xt calls the function during the event-loop phase in response to an occurrence

However, callbacks are often invoked by widgets from within *actions*, which are event-driven. Action routines are called directly by Xt in response to events specified in a *translation table*. The Xt Translation Manager supports a high-level, event-specification syntax, which allows easy specification of complicated event sequences (such as double- or triple-clicks, or key- and button-press combinations) as the trigger for actions. Furthermore, the translation table is a resource, allowing the application developer or the user to configure the events that will invoke a given widget action.

Actions may be internal to the widget and require no interaction with the application. (For example, in the Athena Text widget, all editing operations are carried out entirely by the widget, using functions defined as actions. The only role required of the application is to read and write files.) However, an application can also add actions to a widget, which can function in much the same way as callbacks, but without the widget class having made provision for them.

The purpose of a well-designed widget set is to implement a particular user interface, which provides conventions designed to make all applications operate in the same way. A widget's callbacks are often designed to support the intended use of the widget, while adding actions to a widget can make it behave in ways the designer did not foresee and that the user might not expect. Nonetheless, there are cases in which the best way to implement the desired behavior is to add actions to an existing widget.

In addition to callbacks and actions, it is also possible for an application (or a widget) to implement *event handlers*, which use a specific event-selection mechanism similar to that used in Xlib. Event handlers are rarely used by application programmers, since actions are simpler and more configurable.

Some widgets also declare public routines, which can be used by an application to control aspects of the widget's behavior or to get widget data. Usually, the purpose of public routines is to provide a more convenient means for setting or getting widget data that would otherwise have to be accessed through resources.

2.1.6 Xt and Object-oriented Programming (OOP)

Xt provides an object-oriented programming (OOP) style, where the objects are widgets. However, since Xt is written in C, a language that provides no special support for OOP, Xt depends on programming conventions and programmer discipline to maintain the semblance of objects. It is very important that the programmer understand the goals and rules of OOP, because the language and the system won't enforce these rules. If you are familiar with another object-oriented system, you will need to understand Xt's particular implementation of OOP. On the other hand, if Xt is your first exposure to OOP, an explanation of its goals and concepts should make the whole system make a lot more sense.

Traditionally, object-oriented programming is defined in terms of the five words *object*, *method*, *message*, *class*, and *instance*, and the concept of *encapsulation*. We've already talked about classes and instances. This section describes the remainder of these terms.

2.1.6.1 The Object

In OOP, an object contains two elements: the data that represents a state, and code that reads or writes that data (called methods) and performs some action based on it. For example, the string displayed by a Label widget is part of its state data, and the code that actually draws the string on the window is a method that reads the state data and draws based on it. Inside widget code, the state data is represented as structure members, and the methods are represented as pointers to functions. Some state data members are public; they are resources that can be set or retrieved from outside the object. Other state data members are private; they cannot be read or written from outside.

2.1.6.2 Methods

What is called a method in traditional OOP is either a method or an action in Xt. In Xt, methods are a set of functions that are fixed for a particular class, triggered in fixed ways in response to Xt function calls made by the application (with one special case, the `expose` method, triggered directly by the `Expose` event). A widget's methods supply its most basic functions, such as the code needed to create a widget or to change its resources. Actions, on the other hand, are called in response to the events specified in a translation table, and are thus user configurable. Actions supply most of the features of widgets, and these features can be added to or replaced by the application, as demonstrated in Sections 2.4.2, 4.3, and 4.4.

2.1.6.3 Messages

In pure OOP, input to objects and communication between them are called *messages*. In Xt, however, the forms of communication are function calls, events, actions, and callbacks. As you have seen, applications can communicate with widgets using function calls, such as to set or get widget resources using `XtSetValues` and `XtGetValues`. Widgets also respond directly to events from the user. Widgets contact the application when certain things occur using callbacks or actions. Widgets pass data back and forth using special kinds of events. All these types of communication can be thought of as forms of messages.

2.1.6.4 Encapsulation

Objects are intended to be black boxes with documented inputs and outputs. In other words, a program that uses an object must not depend on the internal implementation of the object, but instead only on the known inputs and outputs. This is called code *encapsulation*. The advantages of code encapsulation are that programmers can use the object without needing to understand its internal implementation (hiding details), and that the internal implementation of the object can be changed at any time because no other code depends on it. This can be stated in another way: it minimizes interdependencies. In large software projects, this one feature makes OOP worthwhile.

This encapsulation is very effective in Xt. You should be able to get a long way toward completing an application without even needing to know what the code inside a widget looks like, let alone the details that implement a particular widget. For that reason, in this book we don't show you what is inside a widget until Chapter 5, *Inside a Widget*. Even when you do know how widgets are implemented, it is a good idea to "forget" while writing application code, so that you are not tempted to depend on implementation details.

Each widget class has a public include file and a private include file. The application includes just the public include file, while the actual widget code includes the private (which happens also to include the public). The names of both include files are based on the class name of the widget, but the private include file adds the letter P to the end. For example, the Label widget's public include file is *Label.h*, while its private include file is *LabelP.h*. Application code should include only the public include file, to maintain the desired encapsulation, It is tempting for beginning Xt programmers to include the private include file, because it allows you to take shortcuts, but you should resist the temptation.*

*In a language designed for OOP, such as C++, these shortcuts are generally prevented by the compiler or by the language itself.

2.2 Structure of X Toolkit Applications

All Toolkit applications have the same basic structure, as follows:

1. Include *<X11/Intrinsic.h>* and *<X11/StringDefs.h>*, the standard header files for Xt.
2. Include the public header file for each widget class used in the application. (Each widget class also has a private header file, which is used in the widget code.)
3. Initialize the Toolkit with the convenience function `XtAppInitialize` or `XtVaAppInitialize`. (These two functions do exactly the same thing but have slightly different argument styles. This will be described in Section 2.5.1. Many other Xt functions also have two versions with and without Va in their names. In the future, unless specifically stated otherwise, you can assume that when a sentence mentions one version it also applies to the other.)
4. Create widgets and tell their composite parent widget about them. This requires one call to `XtVaCreateManagedWidget` for each widget. (Separate `XtVaCreateWidget` and `XtManageChildren` calls can also be used to speed startup of applications with many widgets.)
5. Register callbacks, actions, and event handlers, if any, with Xt.
6. Realize the widgets by calling `XtRealizeWidget`. This function has to be called only once in the entire application, passing it the shell widget returned by `XtVaAppInitialize`. This step actually creates the windows for widgets and maps them on the screen. This step is separate from creating the widgets themselves, because it allows all interdependent widgets to work out their relative size and position before any windows are created. (More on this topic, called *geometry management*, in Chapter 11, *Geometry Management*.)
7. Begin the loop processing events by calling `XtAppMainLoop`. At this point, Xt takes control of your application and operates the widgets. If any widgets are to call functions in your application, you must have registered these functions with Xt before calling `XtAppMainLoop`.

The four steps (3 through 6 above) of initializing the Toolkit, creating widgets, registering application functions with Xt, and realizing widgets, comprise the setup phase of the application. Your application should do as much of its work as possible in this phase, before calling `XtAppMainLoop`, in order to speed up the second phase of the application (which is the event loop). This policy improves the response time to user events.* As we will see in Chapter 5, *Inside a Widget*, this policy also applies to the code inside a widget. It is basic to X, and indeed to any event-driven system.

*In X protocol terms, the setup phase consists of requests to the server that are generally long, and often require immediate replies, which tends to slow performance. The event loop phase, on the other hand, generally consists of short requests and very few "round-trip" requests. For a full description of this concept, see the introductory chapter to Volume Zero, *X Protocol Reference Manual*.

2.3 A Simple X Toolkit Application

Some application code that uses the X Toolkit will go a long way to illustrate the basic concepts introduced above.

Figure 2-11 shows the actual window created by a minimal "hello, world" Toolkit application *xhello*, and Example 2-1 shows the code for it. *xhello* simply displays the string "hello" in a window.

Figure 2-11. xhello: appearance on screen

Most window managers add a decorated border above or surrounding the window; this is not part of the application. Likewise, the placement of the widget on the screen depends on the window manager. Since no coordinates are specified as resources for the widget, most window managers will require the user to place it interactively. Some window managers can be configured to place new windows at an arbitrary default location.

2.3.1 The Code

Example 2-1 shows the code for *xhello.c*.

Example 2-1. xhello.c: a minimal "hello, world" application

```
/*
 * xhello.c - simple program to put up a banner on the display
 */

/*
 * Include files required for all Toolkit programs
 */
#include <X11/Intrinsic.h>   /* Intrinsics Definitions*/
#include <X11/StringDefs.h>  /* Standard Name-String definitions*/

/*
 * Public include file for widgets we actually use in this file.
 */
#include <X11/Xaw/Label.h>   /* Athena Label Widget */

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, hello;
```

Example 2-1. xhello.c: a minimal "hello, world" application (continued)

```
    topLevel = XtVaAppInitialize(
            &app_context,       /* Application context */
            "XHello",           /* Application class */
            NULL, 0,            /* command line option list */
            &argc, argv,        /* command line args */
            NULL,               /* for missing app-defaults file */
            NULL);              /* terminate varargs list */

    hello = XtVaCreateManagedWidget(
            "hello",            /* arbitrary widget name */
            labelWidgetClass,   /* widget class from Label.h */
            topLevel,           /* parent widget */
            NULL);              /* terminate varargs list */

    /*
     *  Create windows for widgets and map them.
     */
    XtRealizeWidget(topLevel);

    /*
     *  Loop for events.
     */
    XtAppMainLoop(app_context);
}
```

Each of the four Xt Intrinsics calls in *xhello* do more than meets the eye:

- `XtVaAppInitialize` performs several important tasks. It reads the resource databases and merges in any command line arguments so that Xt can use this information to configure widgets as they are created. It also opens a connection to the server, and creates a Shell widget that is designed to interact with the window manager and to be the parent for other widgets created in the application.

  The first argument to `XtVaAppInitialize` passes the *address* of an `XtAppContext`. An `XtAppContext` is an opaque pointer to a large structure in which Xt will manage all the data associated with the application. The only use of the `XtAppContext` returned from `XtVaAppInitialize` in a typical application is to pass it to `XtAppMainLoop`. The true purpose for the `XtAppContext` being a public variable is complicated and will be discussed later.

  The second argument (a string) is the class of the application. It is the string that can be used in resource files to set resources for this application, and it is also the name of the application-defaults file, in which the application writer establishes default resource settings for the application. By convention, the class name is the same name as the application name (the string one types to invoke the application), except with the first letter capitalized, or if the application name begins with X, the first two letters capitalized. For the current application, the class name is `XHello` since the application name is *xhello*.

  The remaining arguments have special purposes that are not used in this application. We will introduce them briefly here but reserve complete treatment of them for Chapter 3, *More Techniques for Using Widgets*. The third and fourth arguments are a pointer to and length of an array of application-specific command-line arguments which you can define. The fifth and sixth arguments are the common `argc` and `argv`—which

`XtVaAppInitialize` parses for a variety of standard X Toolkit options and the ones you defined in the previous arguments. The seventh argument is where you specify fallback resource settings in case the app-defaults file is not installed properly. The final argument terminates a variable-length argument list which can be used to customize the Shell widget with resources: this application uses the default Shell widget and therefore provides no resource settings in the list.

`XtVaAppInitialize` removes all arguments it recognizes from *`argc`* and *`argv`*, except for the application name. The program can check if more than one argument remains, and if so, abort with an error message.

The Shell widget returned by the call to `XtVaAppInitialize` is used as the parent of all other widgets in the application (except pop ups, which receive their own pop-up shell parent.)

- The `XtVaCreateManagedWidget` call both creates the Label widget and tells the parent (the Shell widget) that the Label widget's geometry is to be managed. It is also possible to call `XtVaCreateWidget` and either `XtManageChild` or `XtManageChildren` separately, but this is usually done only if you want to create many children of a single widget, then put them all under parental management at once.

  `XtVaCreateManagedWidget` is used for creating any class of widget. The first argument is the instance name, which Xt uses to look up settings in the resource database. The second argument specifies the class of widget to create—this variable comes from the header file for that widget, and should always be found on the reference page for a widget. The third argument is the parent, which in this case is the Shell widget returned by `XtVaAppInitialize`, but in a more complex example could be a composite widget deeper in a hierarchy of nested widgets. The fourth argument terminates a varargs list, unused in this example, that is for hardcoding widget resources. (More on varargs lists later.)

  Notice that `XtVaCreateManagedWidget` and `XtVaAppInitialize` each return a value of type `Widget`. This type is an opaque pointer that is used to refer to a widget instance. It is used anywhere in the application that you need to refer to a particular widget. We sometimes refer to this as a *widget ID*, since even though it is a pointer to a structure, the individual fields in that structure should never be accessed from the application.

- The steps of initializing the Toolkit and creating widgets seem logical. However, what does it mean to realize a widget? `XtRealizeWidget` actually makes windows for the widgets, whereas creating the widgets simply creates and initializes various internal widget data structures.* Creation and realization of widgets are separate because some window geometries cannot be known until all the widgets are created and the composite widgets have determined their geometries. The realization step says "OK, all the widgets are created now; calculate their sizes and positions and make windows for them." However, note that it is acceptable to create additional widgets after calling

*A widget is a client-side object. It has no presence on the server other than as a normal X window. The Xt and widget set libraries running on the client side maintain the data that allow programs to work with the abstraction perceived as a widget.

`XtRealizeWidget` as long as there is a good reason to do so (for example, when it cannot be known whether the widget would be needed until a certain user event arrives).

The realization step also maps all the widget windows. Mapping is an X concept, not something added by Xt. Mapping a window makes it eligible for display. For a window to actually become visible, all its ancestors must be mapped. Since `XtRealizeWidget` is called for the widget created by `XtVaAppInitialize`, which is the ancestor of all widgets in the application, the end result is that the entire application is displayed.*

`XtAppMainLoop` transfers control of the application to Xt. From this point on, Xt drives all the widgets in response to events, and it interacts with the application only at times the application has arranged ahead of time. *xhello* did not arrange any interaction with the widget, so Xt operates the Label widget without returning to *xhello*'s code. Examples 2-3 through 2-8 will demonstrate how to make such arrangements.

Xt programming is event-driven. `XtAppMainLoop` dispatches events to widgets and functions in the order in which they occur. These events can be caused by user actions such as pressing a key or by window system actions such as displaying a new window. This is fundamentally different from procedural programming, where the application is in charge and polls for user input at certain points.

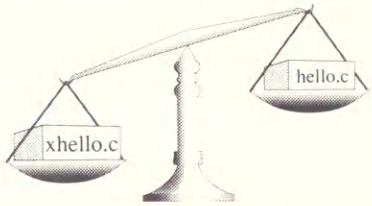
2.3.2 Compiling the Application

You can get the code for *xhello.c* and all the rest of the examples in this book via *uucp* or anonymous *ftp*, as described in the Preface. The code is also on the Release 4 distribution tape from MIT in the "contrib" section. It is a good idea to compile and run each example as it is presented.

The example programs come with Imakefiles that should make building them easy if you have the X source. To do this you need the *imake* program (which should already be in */usr/lib/X11* on your system), and you need the configuration files in *TOP/mit/config*, where TOP is the root of your X sources. An Imakefile is a system-independent makefile that is used by *imake* to generate a Makefile. This is necessary because it is impossible to write a Makefile that will work on all systems. You invoke *imake* using the *ximake.sh* shell script also provided in the X sources. Complete instructions for compiling the examples using *imake* are provided in a *README* file in the example source.

*There are several other conditions that can prevent or delay a window from becoming visible, described in Section 2.2.5 of Volume One, *Xlib Programming Manual* (Section 2.2.4 in the 2nd edition of Volume One). These will not affect your Toolkit application as long as you draw into windows at the right time, using one of the techniques described in Section 4.3 and Section 6.3.

Is the X Toolkit Too Complex?
An editorial aside



Window systems may be simple to use, but they are very complex to program. The first thing that strikes the novice X programmer is how complicated everything is. Learning to program the X Window System, even with the help of the X Toolkit, is a far cry from learning say, the C programming language, where the very first page of the tutorial presents a complete running program. The program itself is trivial, but this, the "hello world" program, has become a tacit benchmark of programmability. The assumption is "if you can't write 'hello world' simply, things are badly designed."

In fact, "hello, world" is a pathological example for X; it is a case where the stylized scaffolding outweighs the functional code. You've just seen an X Toolkit equivalent to "hello world," and it is nearly thirty lines long. But it is node-independent, does device-independent graphics, and can be customized by the user to control what font to use, and what color to use for the border, background, and text.

The "hello world" example is indeed a good benchmark of language complexity, but it is not necessarily a good general benchmark for overall programming complexity. It is an especially poor measure for a system that encompasses a generalized distributed software environment, network communication, and device-independent graphics. Most people write complicated applications, not trivial programs like "hello, world," and it is the ease with which a complicated program can be written that is the true test of a language.

For example, consider the difference in the number of lines of code between "hello world" and a text editor. Kernighan & Ritchie's C Programming Language, *where "hello world" was first introduced, doesn't present an editor. However, Kernighan and Plaugher's* Software Tools, *an equally sacred reference from the same era, presents a text editor that contains well over one thousand lines. Furthermore, the editor is line-oriented, not screen-oriented. By comparison, because of the modular design of the Toolkit, and the development of widgets as reusable user-interface components, a programmer can construct a simple screen editor using the Athena Text widget in about 170 lines of code. This editor is user-configurable, device-independent, and based on a ready-made component that can easily be incorporated into any program needing to provide text-editing capabilities.*

This is not to invite absurd comparisons of incommensurate programming tools, but to emphasize that the complexity of X is the outgrowth of added functionality, not unnecessary convolutions of straightforward algorithms. What really matters is what features are provided, how difficult is it to write the kind of application you want to write, and what performance can be achieved.

—Mark Langley

To compile any of the examples on a UNIX system without using *imake*, use the following command line:

```
cc -o filename filename.c -lXaw -lXmu -lXt -lXext -lX11
```

The order of the libraries is important. Xaw relies on Xmu (miscellaneous utilities), and both rely on Xt. Xaw relies on Xext, the X Consortium extension package which supports non-rectangular windows (if you have a pre-R4 server, applications linked with this library will run but you will not see any non-rectangular windows). Both Xmu and Xt in turn depend on Xlib (the *-lX11* link flag specifies Xlib).

2.3.3 The Application-defaults File

As mentioned above, the resource mechanism allows widgets and applications to be customized. Resources may be declared by either a widget or an application, and can be set from any one of several sources, including a user's resource file, the command line, or an application-specific defaults file.

Each resource of a widget has a default value determined by the widget class that declared the resource. However, in many cases, the application wants a different default value, but still wants the user to be able to change the value of that resource.

The Label widget is a case in point. Example 2-1 (*xhello*) set the default string displayed in the Label widget, "hello," by naming the widget `hello` in the call to `XtVaCreateManagedWidget`. It just so happens that the Label widget uses its widget name as the string to be displayed if no other string has been specified in the resource database. However, this trick doesn't exist for the other resources of Label or of other widgets.

The application can provide a default value for resources by hardcoding them in the application source file using the fallback resources argument of `XtAppInitialize`. However, changing these settings would require recompiling the source. Xt provides a better way.

To provide defaults, applications should always create an "application-defaults" resource file, which on UNIX systems is usually stored in the directory */usr/lib/X11/app-defaults*.* For any application, the name of this file should be the same as the `classname` argument to `XtVaAppInitialize`. By convention, this string is the same as the name of the application, with the first letter capitalized. If the application name begins with X, the first two letters should be capitalized. For *xhello* this is *XHello*.†

Example 2-2 shows the contents of the application-defaults file necessary to make *xhello* display the string "Hello, World!" instead of the default "hello."

*New in R4 is a mechanism that allows you to provide a different application defaults file for each language. This will be described in Chapter 9, *Resource Management and Type Conversion*.

†Note that existing applications do not always follow this latter convention. For example, the application-defaults file for *xmh* is called *Xmh*, not *XMh*.

Example 2-2. XHello: the application-defaults file

```
*hello.label:    Hello, World!
```

The name of the widget instance whose string we are setting is `hello`, and the Label widget's resource that sets the string is `label`. After the colon is the string we want the widget to display. The string should not be quoted. White space after the colon is ignored.

The application-defaults file has the same format as all other resource database files. In brief, there are two types of resources: application resources and widget resources. The syntax for specifying application resources is simple:

application_name.resource_name: *value*

Widget resources are more complicated, since there may be multiple instances of the same widget in an application. As a result, you must specify the name not only of the widget, but a pathname starting with the application name and containing the name of each widget in the widget hierarchy leading to the desired widget. For example, in the application *xhello*, the complete resource specification for the resource called *label* for the Label widget called *hello* would be:

```
xhello.hello.label:  Hello, World!
```

To simplify resource specifications, a wildcard syntax may be used, specifying an asterisk instead of a dot, and omitting one or more terms of the fully qualified name. For example, as we've shown above:

```
*hello.label:  Hello, World!
```

or since there is no other Label widget in the application, even:

```
*label:  Hello, World!
```

One possible source of confusion is that the shell widget instance returned by `XtVaAppInitialize` is not named `xhello` in the call that creates it. Its resource name is the same as the name of the application (specified on the command line). However, its class (`XHello`) is specified in the call that created it.

Note that the Resource Manager provides no error messages, and silently ignores resource specification errors of any kind (including spelling errors), so they can be difficult to track down.

You should also be aware that resources, like widgets, have both instances (usually referred to simply as their "names") and classes. Class names are often the same as the corresponding resource name, but with the first letter of the resource name capitalized. For example, the color resource `foreground` has the class `Foreground`. In addition, there may be other resources (such as the cursor color, border color, and so on) that may also have the class `Foreground`.

A resource class name applies to all instances of the resource. For example, all resources of class `Foreground`, regardless of where in the application or widget they were found, could be set by specifying:

```
*Foreground:  blue
```

Or the foreground resource in all widgets of the Command widget class could be set by using the widget class name in place of the instance name:

```
*Command.foreground:  blue
```

Because resource "instance" names take precedence over class names, however, the developer could override the class setting with a specific setting for an individual widget resource.

Note that all resource specifications in files are given as strings, even though the data required for the resource may be of a different type. Xt automatically converts the string found in the resource database files into the appropriate destination type. For example, colors are specified as color names, which are automatically converted to pixel values used to look up the closest color available in the colormap for the window. These converters are normally defined by Xt, Xmu, or the widget class.

Note also that in resource files, you and users specify the resource name or class as shown above. But in any Xt calls from source code, you always use symbolic constants of the form `XtN`*resourceName* or `XtC`*ClassName* (e.g., `XtNforeground` or `XtCForeground`). We'll explain the reason for that shortly.

The complete list of sources for resource database settings, the precedence rules the resource manager uses for establishing the actual resource value when there are conflicting settings in different database sources, and the mechanics of type conversion, are described in detail in Chapter 9, *Resource Management and Type Conversion*. (See also Chapter 9 in Volume Three, *X Window System User's Guide*.) We'll also be returning to the topic of resources with each new example.

2.3.4 To Hardcode or Not to Hardcode

The resource settings you place in the app-defaults file for your application are, as the name suggests, only defaults. The user can place similar settings in a separate file, and user settings override your app-defaults settings. This allows the user to customize the application.

There are pros and cons to applications that can be radically customized by the user through resources. The pros are quite clear:

- A flexible application will meet the needs of a wider audience.
- Users will have a wide variety of systems on which different colors and fonts will look best.
- Users may have a special job in mind for which the application can be customized.

Some of the cons are that:

- A user may make a mistake in the configuration that makes the application inoperable in some way.
- Documentation and technical support become much more difficult.

A partial solution is to document the default resource specifications and require that users reinstall those default specifications before a technical support person tries to solve their problems. In some cases, dialog boxes can be added that require the user to confirm any irreversible actions.

In any case, you may eventually want to hardcode certain resources in the application so that they are not user configurable. This would be done for resources that, if set incorrectly, would make your application operate in an unsafe fashion. Hardcoding a widget resource is done by placing the resource name and the desired value in the final arguments to `XtVaCreateManagedWidget` or `XtCreateManagedWidget`.

This technique is demonstrated later, because you won't need it until the final stages of releasing your application. While an application is under development, it is better to leave all resources configurable from resource files, because this allows you to change them in the application-defaults file without recompiling the source. When the code is stable and almost ready for release, then it's time to determine which resources need to be hardcoded.

2.4 Connecting Widgets to Application Code

The Toolkit is designed so the code that implements the user interface and the code that implements application features can be kept separate. This is an advantage, because it allows either part to be modified without affecting the other. However, these two pieces of code need to be intimately connected, because the user interface must drive the application code.

This section describes the two basic ways of making this connection using callbacks and actions. As mentioned earlier, the general idea of both techniques is that the application registers functions to be called by Xt in sequence in response to occurrences within certain widgets.

The two techniques differ in the way that the registered function is invoked. For callbacks, the trigger is an abstract occurrence defined by the widget, which may or may not be event related. When this happens, the routines on one of a widget's callback lists is invoked by the widget code, using a call to `XtCallCallbacks` or `XtCallCallbackList`. Actions, on the other hand, are invoked directly by Xt's translation mechanism, as the result of an event combination.

An action is a function that performs one widget feature. A callback is a function that performs one application feature. This distinction should be kept in mind, but is not quite as clear as it sounds, since one widget feature is often the ability to call a function that implements an application feature. Callbacks are often invoked by widgets from within action routines. For example, the Command widget defines a callback resource called `XtNcallback` and several action routines, one of which (`notify`) simply calls the functions registered with the `XtNcallback` resource. This combination of the two mechanisms makes the abstract occurrence that triggers a callback customizable. In other words, when these mechanisms are combined you can configure what user behavior will trigger the callback by using the translation manager which maps events into actions.

The two mechanisms also differ in the way they are added to a widget. The application can add a callback routine to a widget using `XtAddCallback` or `XtAddCallbacks` only if the widget has declared a callback list as a resource. When a widget has a callback resource, it means that the widget writer foresaw that users of the widget would want to have application code called in response to a particular user behavior in that widget. Actions, on the other hand, can be added to a widget using `XtAppAddActions` without the widget's knowledge or consent.

A widget class may have more than one callback resource. For example, the Athena Scrollbar widget defines a `XtNscrollProc` callback resource to be invoked when the first or third buttons are clicked in the scrollbar, a `XtNthumbProc` callback resource to be invoked when the user drags the thumb with the pointer, and a `XtNjumpProc` callback resource when the user clicks the second button. Each of these callbacks indicates a different kind of user behavior, and requires a different kind of movement of the data in the associated window. The callback function registered with each of these callback resources would actually move the data in the various ways.

Widgets define actions in order to implement their basic features in a modular way that allows the event sequences that trigger the actions to be customizable. Applications can also add actions to add features that a widget does not already provide. When working with a widget set such as Motif or OPEN LOOK that has clearly-defined user-interface conventions, adding new behavior to a widget may not be an appropriate thing to do. In addition, using actions is somewhat more complex than using callbacks, and actions have no advantages over callbacks when an appropriate callback is available.

The next two sections describe how to use callbacks and then how to use actions.

2.4.1 Callbacks

To illustrate the use of callbacks from the application, we will write an application that uses the Athena Command widget. This type of widget is also known as a pushbutton; some analogue will be present in every widget set. It contains a string or picture, and executes a command when a pointer button is clicked on it.

A Command widget calls an application function when you press and release the first pointer button (by default) in its window. It also highlights the window border of the Command widget when you move the pointer into it. When you press and hold the pointer button in the Command widget's window, the Command widget redraws the window in reverse video. Moving the held button out of the window resets the widget without executing the command.

The *xgoodbye* program creates a single Command widget. It is very similar to *xhello*, but takes advantage of the callback provided by the Command widget. Clicking on the "Goodbye, Cruel World" button exits the program.

Figure 2-12 shows the window that *xgoodbye* creates if you have installed the suggested application-defaults file (otherwise, it will display "goodbye"). It is suggested that you compile and run *xgoodbye.c*, testing its response to moving the pointer in and out of its window, and clicking the various pointer buttons on its window.

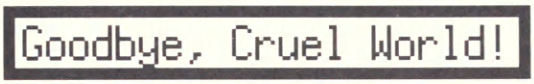


Figure 2-12. The appearance of xgoodbye when the pointer is in the window

This example is not as frivolous as it seems. Many applications use code identical to this to implement their "Quit" button.

The code for *xgoodbye.c* is shown in Example 2-3.

Example 2-3. xgoodbye.c: complete code

```
/*
 * xgoodbye.c - simple program to put up a banner on the display
 *          and callback an application function.
 */

#include <stdio.h>
/*
 * Include files required for all Toolkit programs
 */
#include <X11/Intrinsic.h>      /* Intrinsics Definitions */
#include <X11/StringDefs.h>     /* Standard Name-String definitions */

/*
 * Public include file for widgets we actually use in this file.
 */
#include <X11/Xaw/Command.h>    /* Athena Command Widget */

/*
 * Quit button callback function
 */
/* ARGSUSED */
void Quit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    fprintf(stderr, "It was nice knowing you.\n");
    exit(0);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, goodbye;

    topLevel = XtVaAppInitialize(
            &app_context,       /* Application context */
            "XGoodbye",         /* Application class */
            NULL, 0,            /* command line option list */
            &argc, argv,        /* command line args */
            NULL,               /* for missing app-defaults file */
```

Example 2-3. xgoodbye.c: complete code (continued)

```
            NULL);              /* terminate varargs list */

    goodbye = XtVaCreateManagedWidget(
            "goodbye",          /* arbitrary widget name */
            commandWidgetClass,/* widget class from Command.h */
            topLevel,           /* parent widget*/
            NULL);              /* terminate varargs list */

    XtAddCallback(goodbye, XtNcallback, Quit, 0 /* client_data */);

    /*
     *  Create windows for widgets and map them.
     */
    XtRealizeWidget(topLevel);

    /*
     *  Loop for events.
     */
    XtAppMainLoop(app_context);
}
```

And here is *xgoodbye*'s application-defaults file:

Example 2-4. XGoodbye: the application-defaults file

```
*goodbye.label:     Goodbye, Cruel World!
```

The differences between *xgoodbye* and *xhello* all apply to adding a callback function. In this example we have some application code (the `Quit` function) that we register with Xt as a callback function for the widget called `goodbye` using the `XtAddCallback` call.

The `Quit` function is defined before `main`, so that we can use the function pointer `Quit` in the `XtAddCallback` call. It is also possible to declare `Quit` as a function pointer early in the application, but actually to define it further down in the source code.

The `XtAddCallback` call used to register `Quit` as the Command widget's callback is as follows:

```
XtAddCallback(goodbye, XtNcallback, Quit, 0);
```

The first argument, `goodbye`, is the widget that is to trigger the callback. The second argument, `XtNcallback`, is a symbolic constant that identifies which of the widget's callback resources is being set. (This constant, like most resource names defined by Xt, is defined in the file *<X11/StringDefs.h>*. Additional resource names are often defined by an individual widget's public header file.) In this particular case, the name `XtNcallback` does not shed any light on the real purpose of the callback. A better name would have been `XtNnotifyCallback` or `XtNcommandCallback`. All widgets also have the callback `XtNdestroyCallback`, which is called when the widget is destroyed. Scrollbar widgets often have at least two callbacks in addition to `XtNdestroyCallback`; in the Athena set these are called `XtNscrollProc` and `XtNthumbProc`, and they are called, respectively, when the user clicks in the scroll area or drags the thumb.

The third argument of `XtAddCallback` is the function the widget is to call, and the last argument is any data to be passed to the callback function. No data is to be passed to `Quit`.

The `Quit` function itself, like all callback functions, takes three arguments:

```
void Quit(w, client_data, call_data)
```

- The first argument is the widget that triggered the callback, as specified as the first argument in `XtAddCallback`. You would use the value of this argument in your callback function if you registered the same function as a callback for two different widgets, and if you wanted to distinguish in the callback which widget called it.
- The second argument, *`client_data`*, is the value passed as the last argument of `XtAddCallback`. This can be any data that you need in the callback function.
- The third argument, *`call_data`*, is a piece of data passed from the widget. Some classes of widget set this argument, but others do not. The documentation for the widget will specify the contents of this data if it is used. The Command widget doesn't provide any *`call_data`*, but the Scrollbar widget, for example, passes back the current position of the thumb.

If you are intentionally not going to use one or more of these arguments, you should place the comment `/*ARGSUSED*/` on the line before the function. This prevents the C program checker *lint* from complaining about the unused arguments. See the Nutshell Handbook *Checking C Programs with lint* for more details. By convention, callback function names (and as you will see, action function names as well) are capitalized. This lessens the chance of collision with other variables names because callback functions are global in the source file. If this convention is not used, the most common collision is between callback functions and widget IDs of type `Widget`. It is tempting to call the quit callback function `quit`, and also to call the quit widget `quit`, but this will result in mysterious errors or a core dump. If you follow the convention that callback functions are given a capitalized name, such as `Quit`, you avoid this problem.

2.4.2 Actions

The second way of connecting an application with a widget is using actions. Actions are most appropriate for adding minor features to an existing widget when the widget does not provide a callback that you want, or for building an specialized application window by adding features to a Core widget.

Usually, when a widget does not provide a certain callback, it also does not provide various other characteristics that you want. For example, to make the Label widget work like Command (assuming the Command widget didn't already exist), you would have to make it accept more kinds of input, add the drawing code to highlight the border and draw the text in reverse video, and add the ability to call an application function. All this can be done with actions, but it would take a lot of work. What makes it difficult is that your code may interact with the widget's code in unpleasant ways. When the changes are so major, it makes more sense to create a new widget subclass, which shares some characteristics and code with its superclass. As we will see, that is exactly how Command is implemented, as a subclass of Label. The Core widget, on the other hand, has no input or output semantics at all, and therefore is simpler to add actions to without conflict. We'll demonstrate a simple feature addition here, and demonstrate the use of the Core widget in Chapter 4, *An Example Application*.

A simple example that shows the power of actions is to replace the `Quit` callback function shown in Example 2-2 with two separate quit actions, one of which prompts the user for confirmation, the other of which does not. As separate actions, these functions can be configured to respond to different events. Ideally, we'd like to emulate the Macintosh user-interface semantic, which allows a single click to select a button, which is then confirmed by a carriage return in response to a dialog box, while a double click performs the button's action right away, without confirmation. However, the way the Translation Manager interprets event sequences, you cannot specify both a single and double click for the same widget if you want them to execute mutually exclusive functions (i.e., unless you don't mind the single-click action being executed in addition to the double-click action whenever you double click).

Instead, we'll have the confirm action respond to a click of the first pointer button, and quit without confirmation in response to the second.

Here's the complete code for *xfarewell.c*.

Example 2-5. xfarewell.c: complete code

```
 * xfarewell.c - simple program to provide a Command widget
 * that performs a different action in response to a
 * click of the first and second pointer buttons.
 */

#include <stdio.h>
/*
 * Include files required for all Toolkit programs
 */
#include <X11/Intrinsic.h>          /* Intrinsics Definitions */
#include <X11/StringDefs.h>         /* Standard String definitions */

/*
 * Public include file for widgets we actually use in this file.
 */
#include <X11/Xaw/Command.h>        /* Athena Command Widget */

/*
 *  Confirm action
 */
/*ARGSUSED*/
static void Confirm(w, event, params, num_params)
Widget w;
XButtonEvent *event;
String *params;
Cardinal *num_params;
{
/*
 * Once we show how to do it, we could pop-up a dialog box to do this.
 * Since we haven't yet, simply print a message to stderr.
 */
    fprintf(stderr, "Are you sure you want to exit?\n\
            Click with the middle pointer button if you're sure.\n");
}

/*
 *  Quit action
 */
/*ARGSUSED*/
static void Quit(w, event, params, num_params)
```

Example 2-5. xfarewell.c: complete code (continued)

```
Widget w;
XButtonEvent *event;
String *params;
Cardinal *num_params;
{
    fprintf(stderr, "It was nice knowing you.\n");
    exit(0);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, farewell;

    static XtActionsRec two_quits[] = {
        {"confirm", Confirm},
        {"quit", Quit},
    };

    topLevel = XtVaAppInitialize(
            &app_context,       /* Application context */
            "XFarewell",        /* Application class */
            NULL, 0,            /* command line option list */
            &argc, argv,        /* command line args */
            NULL,               /* for missing app-defaults file */
            NULL);              /* terminate varargs list */

    farewell = XtVaCreateManagedWidget(
            "farewell",         /* arbitrary widget name */
            commandWidgetClass, /* widget class from Command.h */
            topLevel,           /* parent widget*/
            NULL);              /* terminate varargs list */

    XtAppAddActions(app_context, two_quits, XtNumber(two_quits));

    /*
     *  Create windows for widgets and map them.
     */
    XtRealizeWidget(topLevel);

    /*
     *  Loop for events.
     */
    XtAppMainLoop(app_context);
}
```

In Example 2-5 above, we have dispensed entirely with the use of the Command widget's callback. Instead, we have implemented two separate application functions, `Confirm` and `Quit`, that will be registered as actions of the Command widget. To make the actions available, you must declare an actions table, which maps strings that the user can specify in a translation table to function pointers.

You must register the actions table with Xt by calling `XtAppAddActions`. (The `XtNumber` call counts the number of entries defined in the actions table—you will see this macro often.) You should also create a default translation table and store it in the application-defaults file.

2.4.2.1 The Actions Table

The format of an actions table is defined as follows:

```
typedef struct _XtActionsRec{
    char *string;
    XtActionProc proc;
} XtActionsRec;
```

By convention, the string and the function name are identical, except that the function name should begin with an upper-case letter, as in the following example:

```
static XtActionsRec two_quits[] = {
    {"confirm", Confirm},
    {"quit", Quit},
};
```

Action names and action functions should not start with Xt or xt: these action names are reserved for the Intrinsics.

This mapping from strings to function pointers is necessary to allow translation tables to be specified in resource files, which are made up entirely of strings.

2.4.2.2 Format of an Action Procedure

An `XtActionProc` is just a pointer to a function with four arguments: a widget, an event, a string containing any arguments specified for the action in the translation table (described in the next section), and the number of arguments contained in the string. The purpose of the last two arguments will be described later. Example 2-6 shows one of *xfarewell*'s action routines.

Example 2-6. An XtActionProc with widget and event arguments

```
/*ARGSUSED*/
static void Confirm(w, event, params, num_params)
Widget w;
XButtonEvent *event;
String *params;
Cardinal *num_params;
{
/*
 * Once we show how to do it, we can pop up a dialog box to do
 * this.  Since we haven't yet, simply print a message to stderr.
 */
    fprintf(stderr, "Are you sure you want to exit?\n\
        Click with the middle pointer button if you're sure.\n");
}
```

Note that for true ANSI C portability, all four arguments to the action function must be declared, even though many compilers will allow you to leave off trailing arguments that are not referenced in the function. If some of the arguments are not used, you should be sure to include the *lint* comment `/*ARGSUSED*/`.

One major difference between an action function and a callback function is that action functions are called with an event as an argument, while actions do not have the `client_data` or `call_data` arguments present for callback functions. This means the only way to pass application data into an action function is through global variables. The presence of the event argument means that you can use the contents of the event structure in the action function. However, be aware that if you are allowing user configuration of the translation table, an action may be called with different kinds of events. You should at least check the event type in the action routine and print an appropriate message if the user has arranged to call the action with the wrong type of event. We'll show how to do this in Chapter 7, *Events, Translations, and Accelerators*.

2.4.2.3 The Translation Table

Actions defined by a widget class are usable only by all instances of that widget class. Actions added with `XtAppAddActions`, on the other hand, are global to the application context, and therefore usable by any widget instance of any class in the application. But a widget uses either type of action only when configured to do so, using a translation table resource, `XtNtranslations`.

Every widget that has actions also has a default translation table that maps event combinations into those actions. The application can override, augment, or replace this table to make a widget call application-registered actions instead of or in addition to the widget's own actions. Registering actions with `XtAppAddActions` makes them eligible for inclusion in translation tables.

Each line of a translation table maps a sequence of events to a sequence of actions. The entire translation table is simply a string consisting of one or more event specifications in angle-brackets, with optional modifiers, followed by a colon and a function name string defined in an action table. Multiple translations are specified as part of the same string. By convention, the string is continued on several lines, one for each translation, with each line except the last terminated with a linefeed (\n) and a backslash (\).

We'll describe the details of event specification and other aspects of translation table syntax in Chapter 7, *Events, Translations, and Accelerators*. For now, an example should get the point across quite clearly.

The default translations for the Command widget are as follows:

```
<EnterWindow>:      highlight()              \n\
<LeaveWindow>:      reset()                  \n\
<Btn1Down>:         set()                    \n\
<Btn1Up>:           notify() unset()
```

That is to say, when the pointer enters the widget, the widget's `highlight` action will be called (this function darkens the border of the widget); when the pointer leaves the window, the widget will be `reset` to its original state. On button 1 (the left-most pointer button) being pressed, the widget's `set` action displays it in reverse video; on button up, the widget's `notify` action (which simply calls the application function pointed to by the `XtNcallback` resource) will be activated, and the `unset` action will return the widget to its normal appearance. Note that `unset()` and `notify()` are both called on the last line: more than one action can be called in response to a single event or event combination.

Example 2-7 shows the application-defaults file for *xfarewell*, which modifies this default translation table.

Example 2-7. XFarewell: the application-defaults file

```
!
! Core resources
!
! The following two lines don't work, but demonstrate a point
*farewell.x: 100
*farewell.y: 100
!
! Even though it syntactically applies to all widgets in the
! application, and all windows have borders, the borderWidth
! resource is only used by certain widgets.  Command is not one
! of them, so this instruction is ignored.
!
*borderWidth: 10
!
*farewell.width: 200
*farewell.height: 100
*farewell.translations: #override\n\
    <Btn2Down>,<Btn2Up>:    quit()\n\
    <Btn1Down>,<Btn1Up>:    confirm() reset()
!
! Label resources
!
*farewell.foreground: white
*farewell.background: black
*farewell.font: helvetica16bi
*farewell.label:  Click on me.
```

The resource set here is `translations`. (This resource has the same name for all widgets.) As you know, the app-default file contents consist solely of strings, so Xt processes the setting for `translations` into a translation table using what's called a resource converter. The string-to-translation-table resource converter recognizes one of three directives, beginning with # on the first line, which tells how to handle existing translations (either set as widget defaults, or in other resource database files):

- `#replace` (the default if no directive is specified) says simply to replace the old translations with the current table.
- `#augment` says to merge the new translations into the existing translation table, but not to disturb any existing translations. If a translation already exists for a particular event, the conflicting translation specification in a table beginning with `#augment` is ignored.
- `#override` says to merge the new translations into the existing translation table, replacing old values with the current specifications in the event of conflict.

We used `#override` because this allows us to keep the translations for `<EnterWindow>` and `<LeaveWindow>` events in place. In addition, the Command widget's `set` and `unset` actions will also remain in effect for `<Btn1Down>` and `<Btn1Up>`. (For more details on why these aren't overridden by the new translations, see Chapter 7, *Events, Translations, and Accelerators*.)

The translation:

```
<Btn1Down>,<Btn1Up>:  confirm()
```

specifies that the `confirm` action should be called in response to a pair of events, namely a button press followed by a button release, with no other events intervening.

The translations:

```
<Btn2Down>:              set()\n\
<Btn2Down>,<Btn2Up>:     quit()
```

specify that the Command widget's internal `set` action should be invoked by pressing button 2, and that our own `quit` action should be invoked by clicking button 2. Note that we don't bother to bind the Command widget's `unset` action to `<Btn2Up>`, since the application will disappear as a result of the `quit` action. (The `unset` action is still used when the user presses button 2 and then moves the pointer outside the widget before releasing button 2. This is one of Command's default bindings that we have not overridden.) If we were using the widget for any other purpose, we would map `<Btn2Up>` to `unset`, so that the widget was restored to its normal appearance when our own action was completed.

2.4.2.4 Hardcoding Translations

There are cases in which an application may want not only to specify translations in an application-defaults file, but also to hardcode them into the application. When you specify translations only in the application-defaults file, the user has unlimited configurability; if the default translations are deleted or changed beyond recognition, the application may no longer work.

Three Xt functions are used for setting translation tables from the application code:

- `XtParseTranslationTable` is used to compile a string translation table into the opaque internal representation `XtTranslations`. (For translations specified in resource files, this conversion is performed automatically by a resource converter.)
- `XtAugmentTranslations` is used, like the `#augment` directive in a resource file, to nondestructively merge translations into a widget's existing translations.
- `XtOverrideTranslations` is used, like the `#override` directive in a resource file, to destructively merge translations into a widget's existing translations.

Both `XtAugmentTranslations` and `XtOverrideTranslations` take as arguments a widget and a compiled translation table returned by `XtParseTranslationTable`.

There is no function to completely replace a widget's translations; however, you can do this by calling `XtSetValues`, the general routine for setting resources (whose use is demonstrated later in this chapter) to set the value of a widget's `XtNtranslations` resource to a compiled translation table returned by `XtParseTranslationTable`.

To set the same translations specified in the application-defaults file from the application itself, we would have used the following code:

Example 2-8. Code Fragment: specifying translations in the program

```
static char defaultTranslations[] = "#override\n\
        <Btn1Down>,<Btn1Up>:    confirm()\n\
        <Btn2Down>:             set()\n\
        <Btn2Down>,<Btn2Up>:    quit()";

XtTranslations mytranslations;
    .
    .
    .
mytranslations = XtParseTranslationTable(defaultTranslations);
XtOverrideTranslations(farewell, mytranslations);
```

As mentioned earlier, you will find it more convenient to place the translation table in the application-defaults file until the last minute, because this allows changes without recompiling the source.

2.5 More About Resources

You now have read about enough techniques to construct an application that uses widgets for its user interface and connects the widgets to application code. However, in order to really take advantage of any widget class, you have to learn about its resources, so that you can set the desired resources in the application-defaults file.

A class defines its own resources and also inherits the resources of all its superclasses. For example, Command supports not only its own resources but also the resources of its superclasses, Label and Core. The documentation for a widget class may describe only the resources for that class and list the name of the superclass (which you can then look up), or it may list all the resources of that class and all superclasses.

2.5.1 Setting and Getting Resources from the Application

Resources are not just for customization of widgets at application startup. The application can change resources of widgets that have already been created, before or while the application is displayed. The application can also query the value of most resources. This section describes first how to set resources and then how to get them.

Setting resources is perhaps most often used for resetting strings in Label widgets. This can be very useful, but you should be aware that setting a resource from the application wipes out the app-defaults or user-specified value for that same resource (if any), unless you first query the resource, and then set it based on its earlier value. It is especially important not to hard-code strings if you want to be able to change the language used by the application simply by changing app-default files.

There are two parallel sets of functions that set or get resources. The two versions of each function have the same name except that one begins with `XtVa` and the other with `Xt` only. The arguments of both functions pass in or pass out resource name/value pairs, but they do it using a slightly different format of arguments. The `XtVa` style uses an ANSI C varargs list,

which is a NULL-terminated list of resource name/value pairs. The Xt version takes an array of Arg structures (called an rgList) and an array length. Each Arg structure contains a resource name/value pair. We will call these two styles the *varargs* style and the *argList* style. The argList routines were the only interface until R4, and they can still be used. Using the argList form of call is slightly more efficient than the varargs form, since internally the varargs routines simply massage their arguments into the argList forms. However, using the argList form is somewhat more verbose, hard to read, and error-prone, and lacks some of the features supported by the varargs form. The loss of efficiency of the varargs form is probably insignificant unless you depend on maximum speed in setting or getting resources many times in a loop. We use the varargs interface in most of this book, but will also show you how to use the arglist form in the following sections so that you can understand existing applications written using them.

The following sections demonstrate how to set and get resources of an existing widget using the arglist and varargs interfaces. The same form of arguments are used in many other functions that set resources for other purposes (usually while creating various types of widgets). Table 2-1 presents the parallel lists of functions. The final arguments of all of these (many of which have not yet been described) are used in exactly the same manner as is about to be shown.

Table 2-1. Functions that Set Resources: arglist and varargs Counterparts

ArgList	Varargs
XtSetValues	XtVaSetValues
XtGetValues	XtVaGetValues
XtCreateWidget	XtVaCreateWidget
XtCreateManagedWidget	XtVaCreateManagedWidget
XtAppCreateShell	XtVaAppCreateShell
XtGetSubresources	XtVaGetSubresources.
XtGetApplicationResources	XtVaGetApplicationResources
XtCreatePopupShell	XtVaCreatePopupShell
XtSetSubvalues	XtVaSetSubvalues
XtGetSubvalues	XtVaGetSubvalues
XtAppInitialize	XtVaAppInitialize

2.5.1.1 Setting Resources with the Varargs Interfaces

The easiest way to set and get resources is to use the Xt functions XtVaSetValues and XtVaGetValues. Each of these functions takes a widget argument and a variable length list of resource name/value pairs, terminated by NULL. These interfaces are new to R4.

Example 2-9 shows the code needed to change the string of a Label widget—this will work any time after the widget has been created, either before or after the widget is realized.

Example 2-9. Using XtVaSetValues to set a widget resource

```
static String new_label = "Hi there."
    .
    .
    .
XtVaSetValues(w,
        XtNlabel, new_label,          /* resource setting */
        XtNjustify, XtJustifyRight,   /* resource setting */
        NULL);                        /* terminate varargs list */
```

Note that a single `XtVaSetValues` call can set any number of resources of a single widget instance. This example also changes the justification of the string.

The symbolic constant `XtNlabel` is the name of the widget resource. The include file *<X11/StringDefs.h>* and the public include files for each widget class (such as *<X11/Xaw/Label.h>*) contain the resource name constant definitions. All resource names and classes are stored in symbolic constants to improve compile-time checking. If you misspell a symbolic constant, the compiler will note the error. If you misspell a string, on the other hand, the error will go unnoticed by the compiler and go unnoticed at runtime as well, but the resource setting will do nothing. Therefore, all resource names are specified using constants of the form `XtN`*name*, where *name* is the resource name.

Also note that the value specified for each resource setting must be the type expected by the widget for that resource, or an error will occur. In this case, the type is `String`, an Xt type which is defined as `char *`. The `XtNlabel` resource happens to be of type `String`, but many resources have values that are not strings. In `XtVaSetValues` calls (and `XtSetValues` calls described in the next section), conversion from strings to the appropriate type does not happen automatically as it does for settings in resource files. However, you can arrange for it to happen by placing special arguments in the `XtVaSetValues` call, as described in Section 3.7.1. (Note that this is the feature of the varargs interfaces that is not supported by the arglist interfaces.)

2.5.1.2 Setting Resources with the ArgList Interfaces

As mentioned above, the use of the arglist interfaces is less elegant than the varargs interfaces just described, and arglist intervaces lack the ability to use Xt's value conversion mechanism. However, the arglist interfaces were the only interfaces until R4, and therefore are used in most existing applications. Example 2-10 shows how to set resources using `XtSetValues`.

Example 2-10. Using XtSetValues to set a widget resource

```
Arg arg;
static String new_label = "Hi there."
    .
    .
    .
XtSetArg(arg, XtNlabel, new_label);
XtSetValues(w, &arg, 1);
```

The `Arg` type is defined as a structure containing the name and value pair that defines a resource:

```
typedef struct {
    String      name;
    XtArgVal    value;
} Arg, *ArgList;
```

The definition of `XtArgVal` differs depending on architecture—its purpose is precisely to make code portable between architectures with different byte sizes. Its use in application code will be demonstrated in Section sargLists. All resource values (in all varargs and argList calls) are limited to the size of `XtArgVal`, which is the largest of `char *`, `caddr_t`, `long`, `int *`, and `proc *`. This means that for larger pieces of data, a pointer must be used. For example, data of type `float` or `double` must be passed as pointers. The documentation for a widget class that declares a resource with a large value should document the fact that a pointer must be passed rather than the value itself.

`XtSetArg` is a macro that makes it more convenient to set the two members of the `Arg` structure. If desired, you can also set the `Arg` structure members like any other C structure by using the `.` or `->` syntax.

Note that the first member of `Arg` is of type `String`, but should be set to one of the `XtN` constants defined in the header file for the widget whose resources are being set.

`XtSetValues` is the call that actually changes the widget resource. You pass it the widget to be reconfigured, a list of `Arg` structures, and the length of the list. Example 2-9 sets only one resource, so the list length is 1.

`XtSetValues` can set any number of resources of a single widget instance. Example 2-11 shows the code necessary to set two resources. Compare this to Example 2-9 to see how much clearer the varargs interfaces are.

Example 2-11. Code fragment to set multiple resources of a widget

```
static String new_label = "Hi there."
Arg args[3];
int i;
    .
    .
    .
i = 0;
XtSetArg(args[i], XtNwidth, 100);  i++;
XtSetArg(args[i], XtNlabel, new_label);  i++;
XtSetValues(w, args, i);
```

Note that the counter `i` cannot be incremented inside the `XtSetArg` macro, because that macro references its first argument twice. Therefore, the counter is customarily incremented on the end of the same line, so that additional resource settings can easily be added.

2.5.1.3 Getting a Resource Value

It is also useful to be able to get the current value of a widget resource. One use of this is to get the current size of a widget. Another use is in finding out what value the user has specified for a particular resource, so that it can be modified with application data. Because C-language arguments are passed by value, and some resource values are not actual values but pointers to them, it is a little more difficult to get widget values than to set them. Example 2-12 shows how to get a pointer to the current string in a Label widget, using the varargs interface.

Example 2-12. Code fragment to get a widget resource using XtVaGetValues

```
#define MAXLEN 256
        .
        .
        .
    Widget hello;
    char userstring[MAXLEN]; /* memory allocated */
    String p;                /* NOTE - memory for array not allocated */
        .
        .
        .
    /* Label widget named hello created here. */
    XtVaGetValues(hello,
            XtNlabel, &p,
            NULL);

    strcpy(userstring, p);
```

The `XtSetArg` call sets the `XtNlabel` argument in the argument list to a pointer to the application variable `p` (which itself is a pointer), and then `XtGetValues` is called, which sets the pointer `p` to point to the widget's resource. The result is that the application knows the value of `p` and `p` points to the resource.

This code may seem nonintuitive. However, since this method is the same for any resource, you can just remember it or look it up even if you don't understand it.

When you query a resource value you are getting the actual address of the storage that Xt uses to store that value. Therefore, you should usually copy any values you have queried from a widget, because if you yourself later change the value pointed to by your variable, it will have unpredictable effects on the widget. For example, if after calling `XtVaGetValues` you copy a new string into `p`, the Label widget will display the new string. The widget will remain the old width, however, regardless ofthe length of the new string, and if the new string is shorter than the old, the remainder of the old string will still be shown. Furthermore, if you change the same resource with a later call to `XtVaSetValues` elsewhere in the application, the value pointed to by your variable will also be changed even though your code never changed it. In short, don't change a queried resource value—change a copy of it.

Example 2-13 shows how to query a widget resource using the arglist interface, `XtGetValues`.

Example 2-13. Code fragment to get a widget resource using XtGetValues

```
#define MAXLEN 256
        .
        .
        .
    Widget hello;
    Arg arg;
    char userstring[MAXLEN];          /* memory allocated */
    String p;     /* NOTE - memory for array not allocated */
        .
        .
        .
    /* Label widget named hello created here. */
    XtSetArg(arg, XtNlabel, &p);
    XtGetValues(hello, &arg, 1);

    strcpy(userstring, p);
```

Note, however, that some resources are not designed to be queried. For example, translation tables and callbacks are compiled into an internal representation, so it is pointless to try to read them.

2.5.2 Core Resources

All widgets are subclasses of Core. Therefore, the resources of the Core class are available for all widgets. Table 2-2 shows the Core resources.

For each resource, both name and class strings are defined. Class strings are often the same as the corresponding resource name string, but with the prefix `XtC` instead of `XtN`, and with the first letter of the resource name capitalized.

The use of resource classes allows the user or the application programmer to specify that all resources of a particular type be given the specified value.

Table 2-2. Core Resources

Name	Class	Type	Default
XtNx	XtCPosition	Position	0
XtNy	XtCPosition	Position	0
XtNwidth	XtCWidth	Dimension	0
XtNheight	XtCHeight	Dimension	0
XtNscreen	XtCScreen	Pointer	from Display
XtNcolormap	XtCColormap	Pointer	from parent
XtNdepth	XtCDepth	int	from parent
XtNbackground	XtCBackground	Pixel	White
XtNbackgroundPixmap	XtCPixmap	Pixmap	NULL
XtNborderWidth	XtCBorderWidth	Dimension	1

Table 2-2. Core Resources (continued)

Name	Class	Type	Default
XtNborderColor	XtCBorderColor	Pixel	Black
XtNborderPixmap	XtCPixmap	Pixmap	NULL
XtNtranslations	XtCTranslations	XtTranslations	NULL
XtNaccelerators	XtCAccelerators	XtTranslations	NULL
XtNmappedWhenManaged	XtCMappedWhenManaged	Boolean	TRUE
XtNdestroyCallback	XtCCallback	Pointer	NULL
XtNsensitive	XtCSensitive	Boolean	TRUE
XtNancestorSensitive	XtCSensitive	Boolean	TRUE

The fact that size and position are resources of all widgets means that your application-defaults file can control the layout of the application. You should always resize widgets by setting these resources—never resize or move a widget's window using Xlib calls since this would make Xt's knowledge of window geometries inaccurate. Nor should you use the Xt functions `XtConfigureWidget`, `XtMoveWidget`, or `XtResizeWidget` from the application. These routines are intended to be used only by geometry-managing composite widgets.

The `XtNscreen`, `XtNcolormap`, and `XtNdepth` resources hold the default screen, colormap, and depth (the number of bits per pixel used for indexing colors in the colormap). The top-level shell widget gets the screen from the DISPLAY environment variable or *-display* command-line option; other widgets inherit that value and cannot change it. The top-level shell widget gets the root window's depth and default colormap; other widgets inherit their parent's depth and colormap, so unless an intervening widget has set a different depth or colormap, a widget will have the root window's depth and colormap by default. Normally you will set the depth and colormap resources only on Shell widget and only when your application has special color requirements.

The `XtNbackground`, `XtNbackgroundPixmap`, `XtNborderWidth`, `XtNborderColor`, and `XtNborderPixmap` widget resources control the background and border of the window created by a widget. The background and border of a window are maintained by the X server, and setting these resources causes an immediate change in the window on the screen. You can set either a background color or a background pixmap, but not both. Whichever is set later takes priority. The same applies to the border.

A *pixmap* is similar to a window but is off-screen. It is an array of pixels. When used as a background or border, a pixmap is *tiled* by laying out multiple copies of it side by side, for the purpose of patterning. Pixmaps are also used for icon patterns, in drawing, and as a temporary drawing surface later copied to a window.

The `XtNbackground` and `XtNborderColor` resources can be set using either a color name string or a hexadecimal color specification. (See Appendix B, *Specifying Fonts and Colors*, for more information on acceptable forms of color specification.)

The value of the XtNbackgroundPixmap and XtNborderPixmap resources should be a pathname to a file containing the bitmap, or to a bitmap created in your program. On UNIX systems, standard X bitmaps are stored in the directory */usr/include/X11/bitmaps*. See Appendix F in Volume Three, *X Window System User's Guide*, for information on these standard bitmaps.

XtNtranslations is the resource that contains the translation table described in Section 2.4.2 above. As we demonstrated there, by setting this resource you can change the events that trigger a widget's actions or the actions your application has registered. XtNaccelerators contains an accelerator table. (Accelerators are an extended form of translations that allow events in one widget to be bound to actions in another.)

XtNmappedWhenManaged is a resource used by geometry-managing widgets to specify whether a widget should be eligible for display (i.e., mapped to the screen) as soon as it is placed under parental geometry management, or whether this should not happen until some later time. We'll talk more about this concept in Chapter 11, *Geometry Management*.

The XtNdestroyCallback resource, as described in Section 2.4.1 above, lets you provide an application function to be called when a widget is destroyed. This is infrequently used, since the Toolkit normally handles the job of freeing widget data structures and any server resources.

The XtNsensitive resource controls whether a widget responds to user input. This allows you to turn on or off user input in a certain widget at will. For example, if you have a Command widget whose command is not allowed at certain times, you would set XtNsensitive to FALSE during the period the command is not allowed. The Command widget changes its own look to indicate that it is invalid.

The XtNancestorSensitive resource specifies whether the widget's parent (or earlier ancestor) is sensitive. Sensitivity is propagated downward, such that if any ancestor is insensitive, a widget is insensitive. This resource will often be hardcoded by the application, since if set incorrectly by the user, it could make the application inoperable.

2.5.3 Other Inherited Resources

Besides the resources inherited from Core, a widget inherits resources from each of its superclasses. For example, the Command widget used in *xfarewell* inherits resources from its superclass, the Label widget class, and from Label's superclass, Simple. As shown in Table 2-3, these include, in addition to the label itself, a font, a foreground color, spacing above and below the string, and a value specifying how the string should be placed in the widget. (Note that there are other resources not listed.)

Table 2-3. Label Resources

Name	Class	Type	Default
XtNfont	XtCFont	XFontStruct*	XtDefaultFont
XtNforeground	XtCForeground	Pixel	XtDefaultForeground
XtNinternalHeight	XtCHeight	Dimension	2
XtNinternalWidth	XtCWidth	Dimension	4
XtNjustify	XtCJustify	XtJustify	XtJustifyCenter
XtNlabel	XtCLabel	String	NULL

It is a worthwhile exercise to experiment with the resources available to an application through its widgets, even with such a simple example as *xgoodbye* or *xfarewell*. For example, consider the resource settings for *xfarewell* shown in Example 2-14.*

Example 2-14. Alternate resource settings for xfarewell

```
!
! Core resources
!
! The following two lines don't work, but demonstrate a point
*farewell.x: 100
*farewell.y: 100
!
! Even though the following specification syntactically applies to
! all widgets in the application, and all windows have borders, the
! borderWidth resource value is only used by certain widgets.
! Command is not one of them, so this instruction is ignored.
!
*borderWidth: 10
!
*farewell.width: 200
*farewell.height: 100
*farewell.translations: #override\n\
    Shift<Btn1Up>:          quit()\n\
    <Btn1Down>,<Btn1Up>:    confirm()
!
! Label resources
!
*farewell.foreground: blue
*farewell.font: helvetica16b
*farewell.label:  Click on me.
```

Note that an exclamation point (!) in column zero begins a comment line in a resource file. The number sign (#), the standard UNIX comment symbol, should not be used.

These settings can either be placed in an *.Xdefaults* file in your home directory, or you can save them in any file you like and load them into the server using the *xrdb* client, as follows:

```
xrdb -merge resource_file
```

*See Appendix B, *Specifying Fonts and Colors*, for more information on the font resource specification shown in the example.

(If you want to repeat the experiment with different values, you should be aware that once resources are set with *xrdb*, they remain in effect. Subsequent invocations of *xrdb -merge* will replace settings for the same resources, but won't remove any others that were set before. To start with a clean slate, use *xrdb -load* instead. Note, however, that this will replace *all* of your resource settings, including those for other applications. See Chapter 9 of Volume Three, *X Window System User's Guide*, for more information on using *xrdb*. See Chapter 9 of this book for more information on other possible sources of resource settings.)

The window that results when you run *xfarewell* with these resource settings is shown in Figure 2-13.

Figure 2-13. xfarewell run with new resource settings

If you spend some time playing with different resource settings, you will find some unexpected behavior. For instance, setting the value of the `x` and `y` resources has no effect. Regardless of their value, the *xfarewell* application is simply placed at the current pointer position.

The reason for this is that these resources set the widget position relative to its parent, and since the `farewell` widget is a child of an identically sized Shell widget, they are meaningless. In its geometry-management policy, the shell widget ignores the value of these resources.

If instead you use the x and y resources to try to set the position of the application:

```
xfarewell.x:  100
xfarewell.y:  100
```

they are ignored also, for a similar reason. It is customary for the window manager to assert control over the position of the main application window (in this case the Shell widget `farewell`), and take the value of these resources, whether set by the application or by the user, simply as "hints" to the desired behavior. There is no guarantee that the window manager will honor these hints. The application is generally free to move widgets within its own window, but not to move itself. The basic X philosophy is that the user (through the window manager), not the application, should be in control. (Kill the window manager, or run a window manager that honors application position hints, and the resource specifications shown just above will work.)

Likewise, you will find that specifying:

```
*farewell.borderWidth:  10
```

has no effect, while:

```
*borderWidth: 10
```

does. The reason for this behavior is that while the Command widget inherits the `borderWidth` resource, it does nothing with its value. Only certain widget classes use the `borderWidth` resource to set the border width of their window. Just because a resource is inherited by a widget does not mean that the widget's methods do anything to use its value.

Another perhaps surprising fact appears if you set the `foreground` resource to `white` instead of `blue`. No text appears in the window, and it does not highlight itself when you move the mouse inside. However, if you click a button, you will find that the widget is still working. What is happening is that the background and foreground colors are now both white, and therefore there is no contrast with which to see the widget's drawing. The widget is not sophisticated enough to check that its two colors are not the same. (Adding this simple check would be easy, but it is much more difficult to tell whether two colors that are not the same contrast enough.)

The cautionary point is that there may be unexpected interactions between resources in widget code. Like programs in general, widgets tend to do what you say, not what you mean. A well-designed widget will minimize ill effects, but given the amount of customization that is possible, it may take some time to uncover all the possible pitfalls. Unfortunately, the documentation for most existing widgets doesn't always do a good job of explaining how resources are used inside the widget.

3

More Techniques for Using Widgets

This chapter describes how to use some of the more complex widgets found in applications, including composite widgets, constraint widgets, and pop ups. It also describes how to define application resources and command-line options, and how to hardcode the value of widget resources when you create a widget.

In This Chapter:

Using Composite Widgets 65
Setting Resources for an Instance Hierarchy 68
Geometry Management in Practice 69
Using Constraint Widgets 72
Using Pop Ups 75
More About Callbacks 81
Passing Data to Callback Functions 81
Callback Lists 84
Application Resources 85
The Application Data Structure 86
The Resource List 86
Getting the Resources 90
Command-line Options 91
Standard Command-line Options 91
Defining Your Own Command-line Options 93
Preventing User Customization of Widget Resources 97
Using the Varargs Interfaces 97
Using the Argument List Interfaces 99
Another Way to Set Arguments 100
Merging Argument Lists 102
More About Application Contexts 102

3
More Techniques for Using Widgets

The techniques described in Chapter 2, *Introduction to the X Toolkit*, will get you started on the path to writing applications. But there are more tools at your disposal. This chapter describes the following techniques:

- How to use composite widgets to create a hierarchy of widgets in an application. This section describes how to use different kinds of composite widgets so that the application can be resized and still look good. It also shows how to set resources for a widget hierarchy in the application-defaults file.
- How to use pop-up widgets such as dialog boxes and menus.
- How to use callback lists (instead of single callback functions), and how to pass data to callback functions.
- How to make your applications easier for the user to customize by defining application-specific resources and command-line arguments.
- How to hardcode the value of resources when you create a widget.
- How and why you should use application contexts.

3.1 Using Composite Widgets

The examples in Chapter 2, *Introduction to the X Toolkit*, were atypical because they contained only one widget. Because any real application has several widgets, some way of laying them out is needed in case the application is resized. Therefore, the first widget you create after calling `XtAppInitialize` is usually a composite widget, whose job it is to manage the layout of a group of child widgets. The parent of this composite widget is the Shell widget created by `XtAppInitialize`, usually called `topLevel`.

This chapter's first example is a small application, *xbox1*, that creates two Command widgets contained in a Box widget. Figure 3-1 shows how the application looks on the screen.

Example 3-1 shows the code that implements *xbox1*.

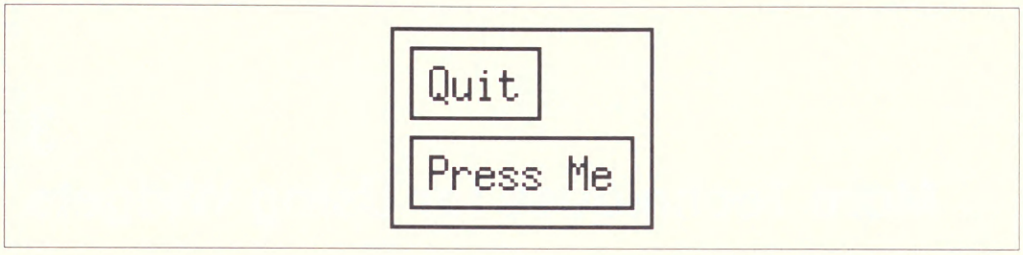


Figure 3-1. xbox1: appearance on the screen

Example 3-1. xbox1.c: complete code

```
/*
 *  xbox1.c - simple button box
 */

/*
 *  So that we can use fprintf:
 */
#include <stdio.h>

/*
 * Standard Toolkit include files:
 */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

/*
 * Public include files for widgets used in this file.
 */
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Box.h>

/*
 * quit button callback function
 */
/*ARGSUSED*/
void Quit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    exit(0);
}

/*
 * "Press me!" button callback function
 */
/*ARGSUSED*/
void PressMe(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    fprintf(stderr, "Thankyou!\n");
}

main(argc, argv)
```

Example 3-1. xbox1.c: complete code (continued)

```
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget box, quit, pressme, topLevel;

    topLevel = XtVaAppInitialize(
            &app_context,           /* Application context */
            "XBox1",                /* Application class */
            NULL, 0,                /* command line option list */
            &argc, argv,            /* command line args */
            NULL,                   /* for missing app-defaults file */
            NULL);                  /* terminate varargs list */

    box = XtVaCreateManagedWidget(
            "box",                  /* widget name */
            boxWidgetClass,         /* widget class */
            topLevel,               /* parent widget*/
            NULL);                  /* terminate varargs list */

    quit = XtVaCreateManagedWidget(
            "quit",                 /* widget name */
            commandWidgetClass,     /* widget class */
            box,                    /* parent widget*/
            NULL);                  /* terminate varargs list */


    pressme = XtVaCreateManagedWidget(
            "pressme",              /* widget name */
            commandWidgetClass,     /* widget class */
            box,                    /* parent widget*/
            NULL);                  /* terminate varargs list */


    XtAddCallback(quit, XtNcallback, Quit, 0);
    XtAddCallback(pressme, XtNcallback, PressMe, 0);

    XtRealizeWidget(topLevel);

    XtAppMainLoop(app_context);
}
```

Example 3-1 creates a Box widget called `box` as a child of `topLevel`, and then creates each Command widget as a child of `box`. Notice how the parent argument of each generation of widgets is used. Also notice that the Shell widget `topLevel` is exactly the same size as `box` and therefore is not visible.

If your application creates many children for a single widget, it may be preferable to create the children with `XtVaCreateWidget`, and then manage all the children of that parent with a single call to `XtManageChildren` (instead of calling `XtVaCreateManaged-Widget` for each child).*

*Just a reminder that all statements about functions such as `XtVaCreateWidget` also apply to the other version of these same functions, such as `XtCreateWidget`, unless specifically stated otherwise.

3.1.1 Setting Resources for an Instance Hierarchy

You have already seen how an application-defaults file can set the string for a Command widget. However, *xbox1* contains two Command widgets, and a widget instance hierarchy that also includes a Box widget. It is worth seeing how to set the Command widget labels in this new situation. (We will be returning often to the subject of setting resources, because it is so important to Toolkit programming. Each time, new ideas will be presented.)

Example 3-2 shows an application-defaults file for *xbox1*.

Example 3-2. XBox1: application-defaults file

```
*pressme*label:     Press Me
*quit*label:    Quit
*Command*background:    green
! The following entry would place the buttons side by side,
! regardless of font.  No setting makes the box widget wider than
! all children side by side or narrower than the widest child.
! *box.width: 1000
```

When an application contains multiple widgets of the same class, resource specifications can either identify individual widget instances by name or can use wildcards or widget class names to reference more than one widget. The first two entries in the example identify the `pressme` and `quit` widgets by instance name. The third entry uses the class name `Command` to set the background of both Command widgets (but not the Box widget) to the color green. This line shows that resources of groups of widgets can be set at the same time. (The second asterisk in each specification is equivalent to a dot, because an asterisk matches zero or more intervening instances or classes.) Because we've used the class name, the background color of all Command widgets in the application, even ones we add later, will be green. To make the Box widget green as well, we could have used an even more general specification such as `*background: green`.

Note that you need to know the instance name for each widget in the application in order to set its resources individually. This is true for all resource files, including the ones customized by the user. Therefore, in the documentation for your application, be sure to include the name and class of each widget in your instance hierarchy. To be thorough, also include a description of the resources of each class, and specify which resources the user can customize.

The first argument of each `XtVaCreateManagedWidget` call is the widget instance name. This name can be used to set resources for this widget in the resource databases. The widget instance name is often the same as that of the variable of type `Widget` that holds the widget ID. This lexical connection is not mandatory, but it is highly recommended because it reduces confusion by helping you to remember the connection between entries in the application-defaults file and the widget instances in the application.

3.1.2 Geometry Management in Practice

Build and run *xbox1*, and then try resizing it to see how the Box widget deals with various geometries. Two possibilities are shown in Figure 3-2.

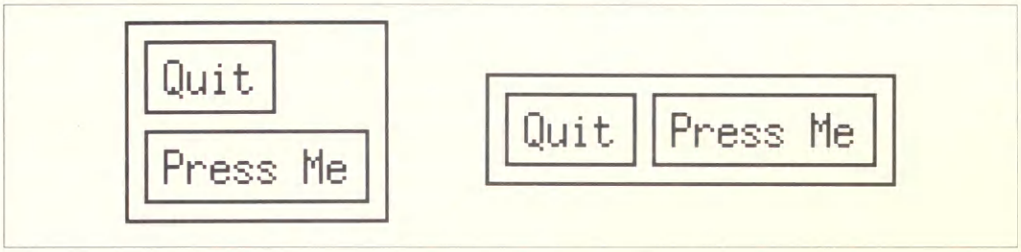


Figure 3-2. Two configurations of xbox1

When you start up the *xbox1* application with its default size, you may notice that the Box widget initially places the widgets one above the other, but upon resizing it places them side by side if there is room. To have the buttons placed side-by-side by default, use resources to set the width of the button box to a value greater than the sum of the widths of the enclosed Command widgets. The commented-out entry in Example 3-2 would do the trick.

Notice also that the effects of resizing from the window manager and resizing by setting the `xbox1.box.width` resource are quite different. Using the window manager, you can resize the box to be smaller than the buttons, causing them to be clipped, or to be far larger than needed, so that they sit at the upper-right corner of a large window. By using resource specifications, however, you can make the box large enough to hold the buttons side-by-side, but no larger. You cannot make it narrower than the width of the widest button, or shorter than the height of a button.

Every widget's size and position is ultimately under the control of the window manager. A box widget attempts to make itself just big enough to hold its children, using the resources provided by the application as a guide, but the window manager can override anything the widget or the application does.

What happens when the user resizes an application is only part of the picture. The application itself may need to resize one of its widgets in order to display more widgets. Or the application may tell a widget to display more data and the widget will have to ask its parent to be resized.

For example, what happens when the application changes the string in one of the Command widgets while the application is displayed?

The Command widget attempts to resize itself to display the current string, by asking its parent for permission. Whether this request is granted depends on the position of the widget in the instance hierarchy, the resizing policy imposed by each composite widget in the hierarchy, and the window manager. This is because each widget, from the Command widget on up, negotiates with its parent when the Command widget requests a new size. If the Command widget already has enough space, it changes the string and no geometry change is necessary. But if the Command widget tries to change size to accommodate the new string

(larger or smaller), the Box widget must approve this change. Since the Box widget is already the same size as the Shell widget, Box can't get any larger without asking Shell. The Shell widget is responsible for negotiating with the window manager. Most window managers will not allow an application to resize itself, because windows should be resized only in response to a user request through the window manager. Therefore, the Box widget will reject the resize request unless it makes the Command widget smaller.

Fortunately, all this negotiation is done by the widgets themselves. The application doesn't need to do anything. However, you should be aware that any widget resource change that results in a size change won't work unless there is enough room in the application for the change to be granted without resizing the top-level window.

Overall, the Box widget is useful for managing Command widgets, because its simple rules work nicely when the children are similarly sized widgets.* However, it can't cope with widgets that are of radically different geometries. Box's decisions about where to place the widgets are inappropriate. Figure 3-3 shows the results upon resizing of a Box widget that is attempting to manage two Scrollbar widgets and a BitmapEdit widget.†

Because geometry management is so complex, there are several different types of composite widgets in most widget sets, each with different rules about how it places children. Many widget sets have a widget specifically designed to place scrollbars next to a main window.

Of course, applications are not limited to using only one composite widget. It is quite common for the application's main window to be a large composite widget which contains several smaller composite widgets, each of which in turn contains certain groups of related widgets. Figure 3-4 shows the *xmh* application and the instance hierarchy used to create it. The top composite widget in *xmh* is of the Athena Paned class. The Paned widget creates several horizontal panels, or panes, one for each child, with a Grip widget positioned on the line between each pane. This Grip is used to change the relative size of the pane. Each pane contains a different functional area of the application. The panes that appear to contain Command widgets actually each contain a single Box, which in turn contains Command widgets. You'll need to design the layout of widgets in your application, decide where in the instance hierarchy to place composite widgets, and experiment to find out which composite widgets provide the best appearance when the application is resized.

One of your tools in arranging child widgets is a subclass of composite widgets called constraint widgets.

*From a user-interface point of view, however, one weakness of any widget like Box is that resizing leaves the Command widgets in new locations where the user may not expect to find them. It is annoying to have to search a box of buttons for the button you want.

†The BitmapEdit widget is not part of the Athena widget set. It is used in Chapter 4, *An Example Application*, and written from scratch in Chapter 5, *Inside a Widget*, and Chapter 6, *Basic Widget Methods*.

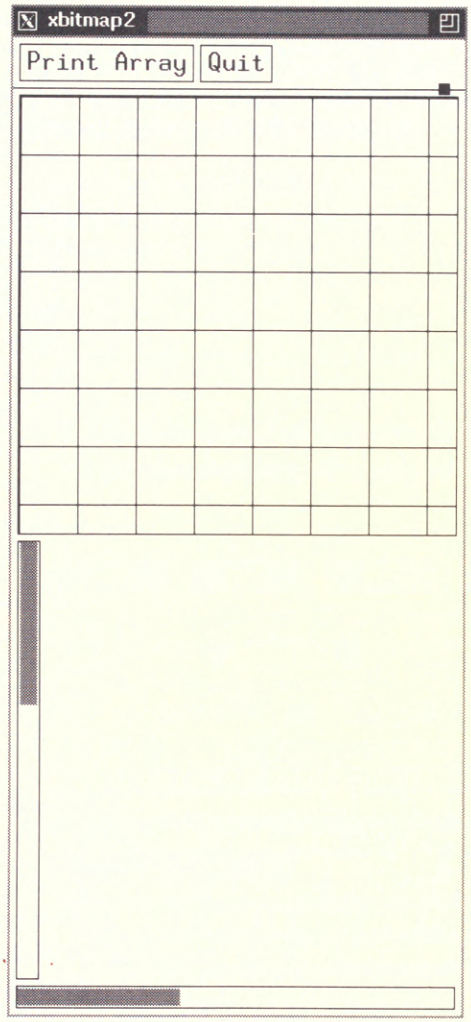


Figure 3-3. Results upon resize of Scrollbar widgets inside a Box widget

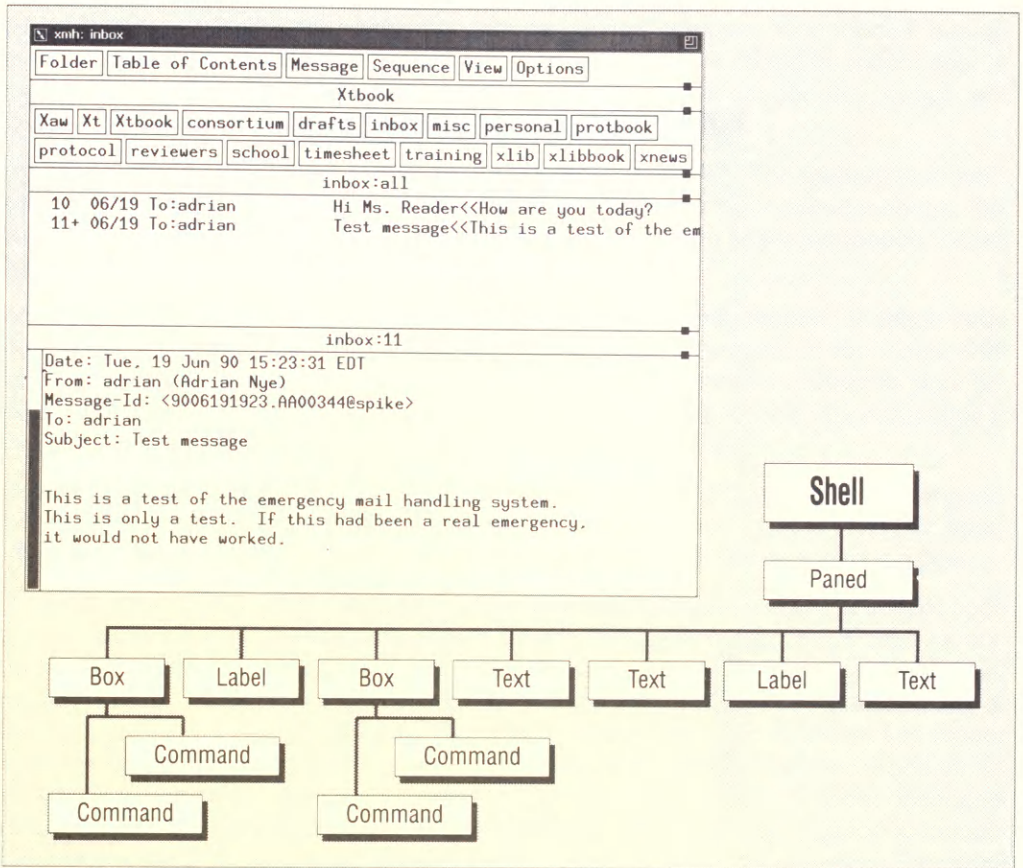


Figure 3-4. The xmh application and its instance hierarchy

3.2 Using Constraint Widgets

Constraint widgets, like composite widgets, manage the layout of children. The difference is that constraint widgets let the application provide layout information for each child. This is a more powerful way to arrange children because it allows you to provide different rules for how each child will be laid out.

In an application, you create a constraint widget just as you would any other widget. But you don't set resources of the constraint widget to specify how each child is laid out. Instead, you set resources of each *child* of the constraint widget. Once any child widget is placed under the management of a constraint widget, the child suddenly has the set of resources defined by the constraint widget, which controls its layout. As usual, these resources can be set from the application-defaults file, from any other resource database file, by hardcoding resources in the application as described in Section 2.3.4, or by using `XtVaSetValues`.

We'll demonstrate how constraint widgets work by replacing the Box widget in *xbox1* (Example 3-1) with an Athena Form widget (a constraint widget). This example is called *xbox2* in the source code. We won't show this code, because the differences from *xbox1* are slight. The only change is that all occurrences of `Box` and `box` are changed to `Form` and `form`, respectively. The real difference between *xbox2* and *xbox1* lies in the setting of resources in the application-defaults file.

The Form widget defines a number of constraint resources, which to the user appear to be resources of the child widgets managed by the Form. Looking at these resources gives you a good idea of the kinds of things that can be done with constraints.

- The resources `XtNhorizDistance` and `XtNfromHoriz` specify the widget position in terms of a specified number of pixels horizontally away from another widget in the form. As an example, `XtNhorizDistance` could equal 10 and `XtNfromHoriz` could be the widget ID of another widget in the Form. (When specified in a resource file, `XtNfromHoriz` is set using the instance name of another widget in the form.) The new widget will always be placed 10 pixels to the right of the widget defined in `XtNfromHoriz`, regardless of the size of the Form. If `XtNfromHoriz` equals `NULL`, then `XtNhorizDistance` is measured from the left edge of the Form.

- Similarly, the resources `XtNvertDistance` and `XtNfromVert` specify the widget position in terms of a specified number of pixels vertically away from another widget in the Form. If `XtNfromVert` equals `NULL`, then `XtNvertDistance` is measured from the top of the Form.

  When set in the application, the values for `XtNfromHoriz` and `XtNFromVert` must be widget IDs. But in the resource database, widget names are used instead, since the actual widget ID changes each time the application is run. This is an example of the automatic type conversion built into the resource manager. Resource conversion is described in Chapter 9, *Resource Management and Type Conversion*.

- The `XtNtop`, `XtNbottom`, `XtNleft`, and `XtNright` resources tell the Form where to position the child when the Form is resized. The values of these resources are specified by the `enum XtEdgeType`, which is defined in *<X11/Form.h>*.

- The values `XtChainTop`, `XtChainBottom`, `XtChainLeft`, and `XtChainRight` specify that a constant distance is to be maintained from an edge of the child to the top, bottom, left, and right edges, respectively, of the Form.

- The value `XtRubber` specifies that a proportional distance from the edge of the child to the left or top edge of the Form is to be maintained when the Form is resized. The proportion is determined from the initial position of the child and the initial size of the Form. Form provides a `StringToEdgeType` conversion to allow the resize constraints to be easily specified in a resource file.

The default width of the Form is the minimum width needed to enclose the children after computing their initial layout, with a margin of `XtNdefaultDistance` at the right and bottom edges. If the Form is assigned a width and height that are too small for the layout, the children will be clipped by the right and bottom edges of the Form.

Example 3-3 shows the application-defaults file for *xbox2*.

Example 3-3. XBox2: application-defaults file

```
*pressme*label:     Press Me
*quit*label:     Quit
*Command*background:     green
*pressme*fromHoriz:     quit
```

Notice that since the instance names are unchanged from *xbox1* (Example 3-1) with the exception that `box` has been changed to `form`, the first three entries in the application-defaults file are also the same. But the next entry, although appearing to affect a resource of the `pressme` Command widget, is actually an instruction for the Form widget about where to place the `pressme` widget relative to the `quit` widget. The effect of this resource setting is shown in Figure 3-5. Note that both the widgets referenced in a constraint must be child widgets of the same constraint widget, in this case the Form widget.

If you run this program you can compare its behavior on resize with the behavior of *xbox1*, and you can experiment with different resource settings for the Form widget.

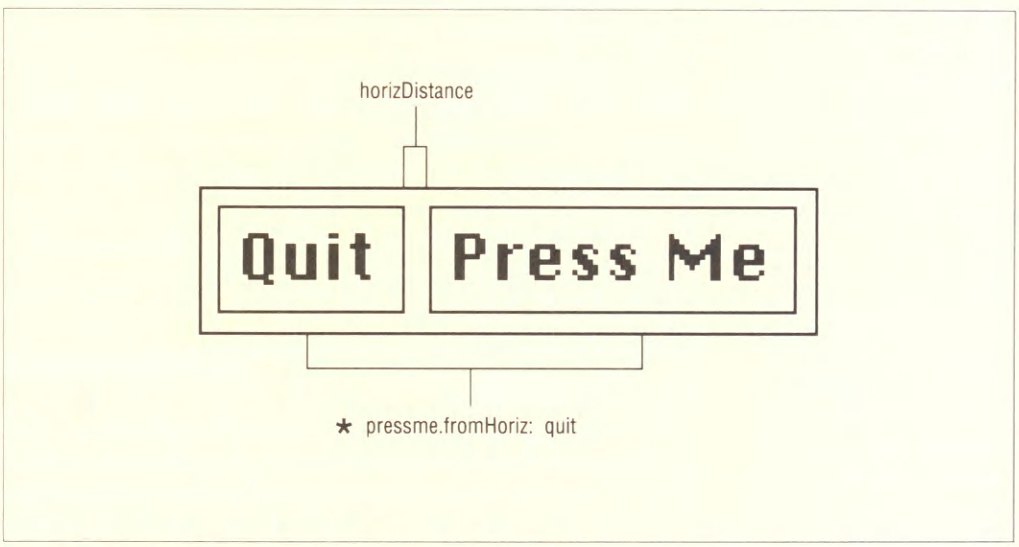


Figure 3-5. Effect of the Form XtNfromHoriz resource

One thing you will notice is that the Form widget is able to resize the Command widgets in addition to or instead of moving them as Box did. This is not a difference between composite and constraint widgets, but simply between Box and Form. Both composite and constraint widgets are allowed to both resize and move their children, but it is up to the design of each widget class whether they actually do so.

Another thing to note is that if you make an error in setting the constraints, it is quite possible to end up with one widget on top of another. Sometimes this can make it appear that one of the widgets has completely disappeared!

3.3 Using Pop Ups

Pop ups are widgets that are not visible until a certain user command is given, or a situation arises in which the program requires user input, and even then are visible for only a short period. The most common examples of pop ups are dialog boxes and menus. In general, a pop up gets information from the user and then goes away.

Pop ups are not a kind of widget, but rather a way of using widgets. Any widget can be used as a pop up. You first create a special parent widget called a TransientShell or a OverrideShell as a child of `topLevel`. Then you create the widget to appear in the pop up, which may be a simple widget or a composite widget with children. However, the TransientShell or OverrideShell widget, like the `topLevel` widget created by `XtVaAppInitialize`, must have only one child.

This process sets up the pop-up widget, but does not put it on the screen. Somewhere in your code you need to call `XtPopup` to pop up the widget and `XtPopdown` when you want to make it invisible again (or use one of a number of other mechanisms for popping up or down widgets). This is typically done in callback or action routines.

The *xbox3* application shown in Example 3-5 adds a pop-up dialog box to *xbox1*. The actual widget popped up is an Athena Dialog widget. The Dialog widget is a widget designed to prompt for auxiliary input from a user. For example, you can use a Dialog widget when an application requires the user to enter some information such as a file name. A Dialog widget is actually just a special case of the Form widget. It provides a convenient way to create a "preconfigured form" useful for dialog boxes.

The typical Dialog widget contains three areas. The first line contains a Label widget providing a description of the function of the Dialog widget, for example, the string ''Filename:''. The second line contains a Text widget into which the user types input. The third line can contain one or more Command widgets that let the user confirm or cancel the Dialog input. (Other types of dialogs may just provide buttons to choose from.) The class variable for the Dialog widget is `dialogWidgetClass`.

Figure 3-6 shows the appearance of *xbox3* when the Dialog widget is popped up.

Note that without proper resource settings in the application-defaults file, the text entry widget will not appear, and the buttons will have different text in them. Example 3-4 shows the required application-defaults file.

Example 3-4. xbox3: application-defaults file

```
*value:
*pressme*label:     Press Me
*quit*label:    Quit
*dialog*label:     Enter Text Below:
*dialog*Command*label:     Dialog Done
```

The `*value:` resource setting tells the Dialog widget to provide a text entry widget but to give it no initial text.

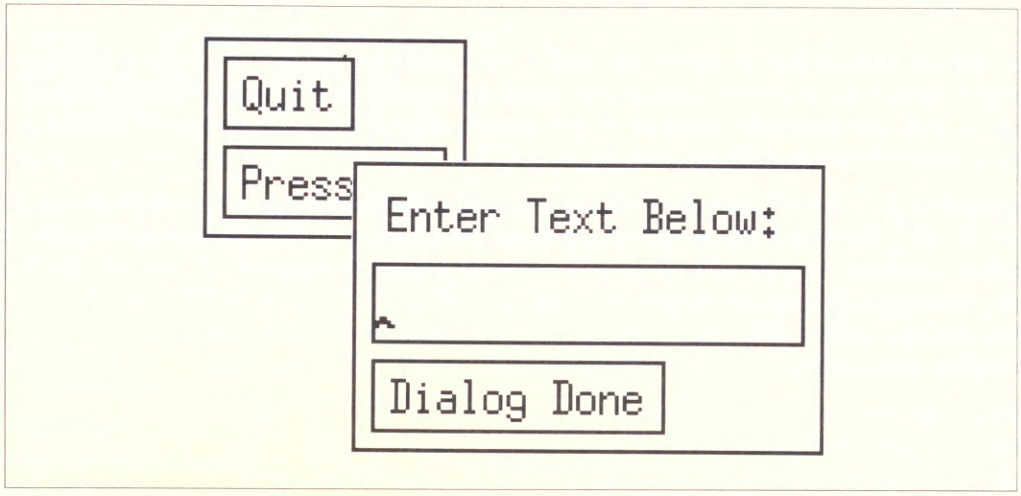


Figure 3-6. xbox3: popping up a Dialog widget

Expanding *xbox* to create *xbox3* requires five steps:

1. Creating the pop-up shell widget.
2. Creating the Dialog widget.
3. Creating a Command widget as a child of the Dialog widget. (This Command widget is used to pop down the dialog box.)
4. Changing the callback function invoked by the "Press Me" button so that it pops up the widget.
5. Writing the callback that pops down the widget.

Example 3-5 shows the code added to *xbox1* to develop *xbox3*.

Example 3-5. Creating a pop-up dialog box

```
     .
     .
     .
#include <X11/Shell.h>

/*
 * Public include files for widgets used in this file.
 */
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Dialog.h>

/*
 * Both dialog and pshell are needed in the dialogDone callback,
 * and both can't be passed in without creating a structure.
 * So we chose to pass dialog in, and make pshell global.
 * pressme and quit are needed in both callbacks, so they are
```

Example 3-5. Creating a pop-up dialog box (continued)

```
 * declared global as well.
 */
Widget pshell, pressme, quit;

/*
 * dialog button
 */
void PopupDialog(w, client_data, call_data)
Widget w;
XtPointer client_data; /* cast to topLevel */
XtPointer call_data;
{
    Widget topLevel = (Widget) client_data;
    Position x, y;
    Dimension width, height;
    Arg arg[2];
    int i;

    /*
      * get the coordinates of the middle of topLevel widget.
      */
    XtVaGetValues(topLevel,
            XtNwidth, &width,
            XtNheight, &height,
            NULL);

    /*
     * translate coordinates in application top-level window
     * into coordinates from root window origin.
     */
    XtTranslateCoords(topLevel,         /* Widget */
            (Position) width/2,         /* x */
            (Position) height/2,        /* y */
            &x, &y);                    /* coords on root window */

    /* move popup shell to this position (it's not visible yet) */
    XtVaSetValues(pshell,
            XtNx, x,
            XtNy, y,
            NULL);

    /*
     * Indicate to user that no other application functions are
     * valid while dialog is popped up...
     */
    XtSetSensitive(pressme, FALSE);

    XtPopup(pshell, XtGrabNonexclusive);
}

/*
 * dialog done button
 */
void DialogDone(w, client_data, call_data)
Widget w;
XtPointer client_data; /* cast to dialog */
XtPointer call_data;
{
```

Example 3-5. Creating a pop-up dialog box (continued)

```
    Widget dialog = (Widget) client_data;
    String string;
    XtPopdown(pshell);
    XtSetSensitive(pressme, TRUE);
    string = XawDialogGetValueString(dialog);
    printf("User typed: %s\n", string);
}
/*
 * quit button callback function
 */
void Quit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    exit(0);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget box, topLevel, dialog, dialogDone;

    /* XtAppInitialize, create box, quit, etc. */

    pressme = XtVaCreateManagedWidget(
            "pressme",                  /* widget name     */
            commandWidgetClass,         /* widget class */
            box,                        /* parent widget*/
            NULL);                      /* terminate varargs list */

    pshell = XtVaCreatePopupShell(
            "pshell",                   /* widget name     */
            transientShellWidgetClass,  /* widget class */
            topLevel,                   /* parent widget*/
            NULL);                      /* terminate varargs list */

    dialog = XtVaCreateManagedWidget(
            "dialog",
            dialogWidgetClass,
            pshell,
            NULL);

    dialogDone = XtVaCreateManagedWidget(
            "dialogDone",
            commandWidgetClass,
            dialog,
            NULL);

    /* callback for quiting application */
    XtAddCallback(quit, XtNcallback, Quit, 0);

    /* callback for popping down dialog */
    XtAddCallback(dialogDone, XtNcallback, DialogDone, dialog);

    /* callback for popping up dialog */
```

Example 3-5. Creating a pop-up dialog box (continued)

```
    XtAddCallback(pressme, XtNcallback, PopupDialog, topLevel);
    XtRealizeWidget(topLevel);
    XtAppMainLoop(app_context);
}
```

The pop-up shell widget is created by a call to `XtVaCreatePopupShell` rather than `XtVaCreateManagedWidget`. The arguments of these two functions are the same, however. The widget class is specified as `transientShellWidgetClass` for dialog boxes, and `overrideShellWidgetClass` for menus. The difference between these two classes is that they are treated differently by the window manager, as will be described further in Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*.

Note that you must include *<X11/Shell.h>* to create a TransientShell or OverrideShell widget. The Shell class actually has several subclasses, each with slightly different characteristics. For convenience, all of Shell's subclasses use the same header file. The subclasses of Shell other than ApplicationShell (returned by `XtVaAppInitialize`), TransientShell (used for dialog boxes), and OverrideShell (used for menus), are described in Section 13.7.

It is often helpful for some or all of the widget IDs involved in pop ups to be global variables in the application. As we will see in the discussion of passing data to callback functions, it is awkward to pass more than one piece of data into any one callback function (one has to create a structure containing them and pass the pointer to the structure). When two or more widget IDs are used in any callback, it is simplest to declare one or more of them global. In Example 3-5, `pshell`, `pressme`, and `quit` are declared global.

The Dialog widget is a *compound* widget—it creates its own widget children in addition to letting the application add children. Its purpose is to make it easier to combine existing widgets in a standard, useful way. It is really a subclass of the Form widget, to which a Label widget child is added to tell the user the purpose of the box and, with certain resource values, an Athena Text widget child is added for text entry.

Ideally, the Text widget should provide a callback (which could be used to provide the data the user entered and popdown the dialog box) when the user types the Return key. Many applications add this missing feature by overriding the translation for the Return key in the Text widget, or by displaying a Command button to provide the callback.

The Label widget and Text widget are automatically created as part of the Dialog widget. You have to explicitly create the Command widget as a child of the Dialog widget. The Dialog widget does, however, automatically set the constraint resources that Form uses to place and size the Command widget. The callback function called by this Command widget is called `DialogDone`.

The `pressme` Command widget's callback function is called `PopupDialog`. Most of its code places the pop-up widget, because pop ups appear at the top left corner of the screen by default. This example centers the corner of the pop up in the top level widget of the application. Since pop up coordinates are relative to the corner of the root window, centering is a three-step process:

1. Get the width and height of top-level window with `XtGetValues`.
2. Translate the center point into root window coordinates with `XtTranslateCoords`.
3. Move the pop-up shell by setting `XtNx` and `XtNy` resources with `XtVaSetValues`.

When you pop up a widget from an action instead of a callback, you can use the content of the event to place the pop up and avoid the last two steps.

Pop-up windows appear at the corner of the screen by default because their windows are children of the root window, not children of the top-level window of the application. This allows the pop up to extend beyond the border of the application, as was shown in Figure 3-5. Pop-up windows are the only difference between the widget hierarchy created by an application and its X window hierarchy. Figure 3-7 shows the two hierarchies for *xbox3*.

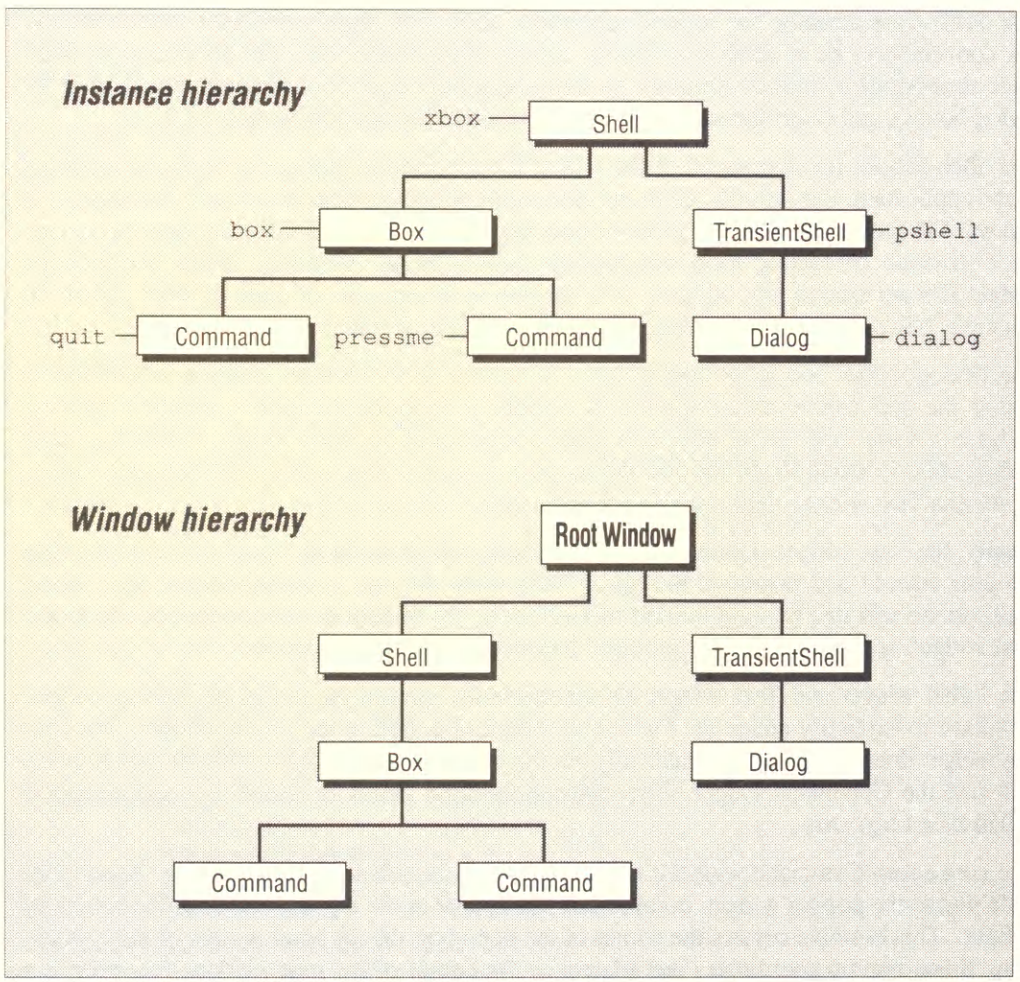


Figure 3-7. The widget instance and X window hierarchies are different only for pop ups

The `PopupDialog` callback function sets the Command widgets in the application to insensitive mode with `XtSetSensitive`* just before it pops up the dialog box. This is done to indicate to the user which Command invoked the current pop up (and that it is futile to press that particular Command again now). When the data is furnished, the application must turn sensitivity back on again.

The `DialogDone` callback function pops down the pop-up shell, sets the application's Command widget back to sensitive mode, gets the value the user typed, and prints it.

If you have a pop-up widget that might never be used, or whose characteristics are not known until just before it is popped up, you can create the pop-up shell and widget just before popping it up, in the callback function or action that pops up the widget. This technique makes the startup time of the application marginally faster, but slows the pop-up time the first time the pop up is used.

Pop ups, and menus in particular, will be described in much more detail in Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*.

3.4 More About Callbacks

As you may recall, a callback is a function that your application wants called in response to a certain occurrence in a certain widget. The application simply declares the callback function and then calls `XtAddCallback`. While Chapter 2, *Introduction to the X Toolkit*, discussed the concept of callbacks and demonstrated the most common use, it did not completely describe all the useful tricks. You can pass application data to callback functions. You can arrange for more than one callback function to be called (in a particular order) when the callback is triggered, and add and remove functions from this list at will. You can declare callbacks statically using a callback list instead of calling `XtAddCallback`.

3.4.1 Passing Data to Callback Functions

A callback function is called with three arguments: *`widget`*, *`client_data`*, and *`call_data`*.

The application determines the use of the *`client_data`* argument, while the widget determines the use of the *`call_data`* argument. As described earlier, the *`widget`* argument is necessary if you have registered the same callback function for two widgets, and you need to know which widget called the function. It also may be used in routine ways, such as for the argument of macros. The *`call_data`* argument is passed in from the widget itself. This argument's value is defined by the widget, and is described in the documentation for the widget unless it is not used. Few of the Athena Widgets use the *`call_data`* argument, but other widgets do use it occasionally.

*This could also be done by setting the widgets' `XtNsensitive` resource with `XtSetValues`. Calling `XtSetSensitive` is slightly faster.

You may pass a single piece of data as the *client_data* argument, or pass a pointer to a structure containing several pieces of data. You may ask, "Why bother with passing data to a callback when I can just declare a global variable?" For one thing, it is a general principle of good C coding to use a global variable only where the variable is needed in several functions and the arguments would otherwise prove unwieldy. Secondly, using the *client_data* argument makes clear the purpose of your variable, whereas it is difficult to trace where a global variable is set or referenced. However, if you find it necessary to change the *client_data* argument in a number of functions in the application, you will need a global variable anyway, and there is nothing that says you must use *client_data* (but be sure to document what you are doing!).

Example 3-6 demonstrates how to pass a single piece of data into a callback function.

Example 3-6. Passing a single value to a callback function

```
/*ARGSUSED*/
void PressMe(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    fprintf(stderr, "%s\n", client_data);
}

main(argc, argv)
int argc;
char **argv;
{
    Widget pressme;
      .
      .
    /* XtAppInitialize, create pressme widget */
      .
      .
    /* last argument is client_data */
    XtAddCallback(pressme, XtNcallback, PressMe, "Thanks");
      .
      .
}
```

Example 3-7 demonstrates passing a pointer to a structure into a callback function. `XtPointer` is a generic pointer type with an implementation-dependent definition. It is used just like the standard C-type `caddr_t`.

Example 3-7. Passing a pointer to a structure to a callback function

```
typedef struct {
    String name;
    String street;
    String city;
} app_stuff;

/*ARGSUSED*/
void PressMe(w, client_data, call_data)
Widget w;
XtPointer client_data;   /* to be cast to app_stuff */
```

Example 3-7. Passing a pointer to a structure to a callback function (continued)

```
XtPointer call_data;
{
    /* cast required in ANSI C */
    app_stuff *address = (app_stuff) client_data;

    fprintf(stderr, "%s\n%s82s0, address->name, address->street,
            address->city);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget box, quit, pressme, topLevel;

    static app_stuff stuff = {
        "John Doe",
        "1776 Constitution Way",
        "Philadelphia, PA 01029"
    };
      .
      .

    /* XtVaAppInitialize, create pressme widget */
      .
      .
    XtAddCallback(pressme, XtNcallback, PressMe, &stuff);
      .
      .
}
```

Note that two coding conventions are required by ANSI C and should be followed, even though many compilers will not complain if they are not used. The first is that all three arguments of the callback must be declared, even if the trailing ones are not used. Many compilers will work fine if unused trailing arguments are omitted from the definition of the callback function. The second is that many compilers will automatically cast `XtPointer` to the type you want if you declare the callback function arguments using that type. The cast on the first line of the `PressMe` callback is required by ANSI C. Example 3-8 shows a definition of the above callback function that works on many compilers but is not ANSI C conformant. Compare Example 3-8 with Example 3-7.

Example 3-8. ANSI non-conformant callback function definition

```
/*ARGSUSED*/
void PressMe(w, address /* call_data omitted */)
Widget w;
app_stuff *address;      /* ANSI non-conformant cast */
/* third arg omitted - ANSI non-conformant */
{
    fprintf(stderr, "%s\n%s83s0, address->name, address->street,
}
```

3.4.2 Callback Lists

You may register any number of callback functions for a single widget callback resource. In other words, when a callback is triggered by a user event, all the functions in the current callback list for that callback for that widget instance will be called, one at a time, in the specified order. Multiple callback functions are not needed in many applications, but can be useful if applied creatively.

Remember that a widget may also have more than one callback list, each triggered by a different occurrence. For example, the Command widget has `XtNcallback`, which is triggered by a pointer button click, and `XtNdestroyCallback`, which is triggered when the widget is destroyed. Each of these callback resources exists as a separate callback list for every instance of the widget.

Xt supplies three standard callback functions that pop up a widget: `XtCallbackNone`, `XtCallbackExclusive`, and `XtCallbackNonexclusive`.*

Do not be misled by the names of the standard pop-up callback functions. Even though their names do not include the word Popup, their only use is to pop up widgets.

Each of these functions pops up the widget specified in the *`client_data`* argument. `XtCallbackExclusive` and `XtCallbackNonexclusive` also *grab* certain events so that the user *must* provide the information requested before continuing. `Exclusive` is for use when you create a single pop up such as a menu or dialog box, and `Nonexclusive` is for when you pop up a menu that pops up submenus. Since these standard pop-up callback functions do not move the pop-up widget before popping it up, it is a good idea to add another callback function to the callback list before them, to place the widget. *xbox5*, which is not shown in this book but is included in the example source, creates a callback list composed of `PrepareDialog` (which moves the pop up to the proper location) and `XtCallbackExclusive`.

Let's take another example of how a callback list might be used. Perhaps you have the functions *A*, *B*, and *C*, and you need to be able to call them separately in response to events in three different Command widgets, or to call all of them in order in response to events in a fourth Command widget. This fourth Command widget might have a callback list including the functions *A*, *B*, and *C*. While the fourth Command widget's code can also be implemented by creating a function *D* that calls *A*, *B*, and *C* and then registering *D* as the callback of the fourth Command widget, you can't pass a different piece of data into each routine, and you can't change the order of the functions called by *D*. You can do both easily if *A*, *B*, and *C* are specified in a callback list. The order in which the callback functions are called can be changed by removing them from the list (using `XtRemoveCallback`, `XtRemoveCallbacks`, or `XtRemoveAllCallbacks`) and then adding them in a different order. You can also call the same function more than once in a callback list.

One way to add more than one callback function is to call `XtAddCallback` more than once. Another way is to call `XtAddCallbacks`, which takes an `XtCallbackRec` array as an argument. This array is usually initialized at compile time, as shown in Example 3-9.

*These standard callback functions also set the calling widget (usually a Command widget) to insensitive mode. You must set the parent widget back to sensitive mode when you pop down the widget.

The final NULL, NULL entry terminates the list. (This particular list registers the functions do_A and then_B, and passes them both 0 as *client_data*.)

Example 3-9. Initializing a callback list

```
XtCallbackRec quit_callback_list[]={
    {do_A, 0},
    {then_B, 0},
    {(XtCallbackProc) NULL, (XtPointer) NULL}
};
```

This form of XtCallbackRec list can also be used to replace a callback list with XtSetValues (but not to *get* a callback list, because Xt compiles the list into an internal form). As demonstrated in Section 3.7.1, an XtCallbackRec list can also be used to register one or more callbacks when creating a widget.

3.5 Application Resources

You already know that widgets declare resources that can be configured through the resource mechanism. In addition, the application itself may have variables that it wants the user to be able to configure from the resource databases, even though these variables have nothing to do with widgets. These are called *application resources*.

Application resources are just like widget resources except that they apply to application code, not to the widgets it creates.

There are three steps for adding application resources. You must:

1. Create an application data structure containing the variables to be set via the resource mechanism.
2. Create a resource table defining the type and default value for each variable.
3. Call XtGetApplicationResources with pointers to the application data structure and resource table as arguments. When this function returns, the application data structure will contain the current settings from the resource databases.

To demonstrate how to get application resources, we will jump ahead and describe a portion of the code for the *xbitmap4* bitmap editor example described in Chapter 4, *An Example Application*. Because this application draws into a widget using Xlib, it needs two colors with which to draw. The bitmap editor also allows the user to specify the dimensions of the bitmap in cells, and the size of each cell in pixels. And, for general utility, it includes a debug flag that can be set in the application to invoke or ignore debugging code without recompiling.

3.5.1 The Application Data Structure

The structure type that contains all the application variables to be set through resources is commonly called `AppData`. Once this structure type is declared, memory can be allocated for a structure called `app_data`. Example 3-10 shows the code that defines the structure type and then allocates memory for the actual structure.

Example 3-10. xbitmap: getting application resources

```
typedef struct {
    Pixel copy_fg;
    Pixel copy_bg;
    int pixmap_width_in_cells;
    int pixmap_height_in_cells;
    int cell_size_in_pixels;
    Boolean debug;
} AppData, *AppDataPtr;

AppData app_data;
```

Note that a pointer called `AppDataPtr` is defined for convenience in defining the resource list (as described in a moment).

As usual in C, the members of the `app_data` structure will be referenced throughout the application code using the dot format. For example, the value of the `debug` field of `app_data` will be referenced with `app_data.debug`.

3.5.2 The Resource List

The resource list looks complicated, but it is easy to understand and even easier to write because it conforms to a consistent pattern. Each field in the application data structure has an entry in the resource list. Each resource user entry in turn has seven fields, which describe the name of the resource, its type and default value, and various other information.

The resource list controls Xt's value conversion facilities. Since user resources are always strings, and application data structure fields can be any type, a conversion may have to take place. Xt has built-in converters to convert from string to most common types needed by applications. These types are called *representation types* by Xt, and they are indicated by constants starting with `XtR`. The representation type of a string is `XtRString`. You control the conversion simply by specifying a resource as a certain representation type in the resource list.

Note that a representation type is different from a C-language type. For example, a color may be represented as an ASCII color name such as "blue," as a structure containing Red, Green, and Blue values, or as a pixel value (an index into a colormap.) It is also possible for two different representations of something to both use the same C-language type. For example, two hypothetical representation types might be `XtRInch` and `XtRMeter`. Both represent distances and both would probably be floating point numbers, but each would have a different value for the same distance.

See Chapter 9, *Resource Management and Type Conversion*, for a description of the standard representation types, as well as information on how to write your own converter routine.

Example 3-11 shows the resource list for *xbitmap*, followed by a description of each of the fields in each entry.

Example 3-11. The resource list for xbitmap

```
/*
 * The following could be placed in a "xbitmap.h" file.
 */
#define XtNdebug "debug"
#define XtCDebug "Debug"
#define XtNpixmapWidthInCells "pixmapWidthInCells"
#define XtCPixmapWidthInCells "PixmapWidthInCells"
#define XtNpixmapHeightInCells "pixmapHeightInCells"
#define XtCPixmapHeightInCells "PixmapHeightInCells"
#define XtNcellSizeInPixels "cellSizeInPixels"
#define XtCCellSizeInPixels "CellSizeInPixels"

static XtResource resources[] = {
    {
        XtNforeground,
        XtCForeground,
        XtRPixel,
        sizeof(Pixel),
        XtOffset(AppDataPtr, copy_fg),
        XtRString,
        XtDefaultForeground
    },
    {
        XtNbackground,
        XtCBackground,
        XtRPixel,
        sizeof(Pixel),
        XtOffset(AppDataPtr, copy_bg),
        XtRString,
        XtDefaultBackground
    },
    {
        XtNpixmapWidthInCells,
        XtCPixmapWidthInCells,
        XtRInt,
        sizeof(int),
        XtOffset(AppDataPtr, pixmap_width_in_cells),
        XtRImmediate,
        (XtPointer) 32,
    },
    {
        XtNpixmapHeightInCells,
        XtCPixmapHeightInCells,
        XtRInt,
        sizeof(int),
        XtOffset(AppDataPtr, pixmap_height_in_cells),
        XtRImmediate,
        (XtPointer) 32,
    },
    {
```

Example 3-11. The resource list for xbitmap (continued)

```
        XtNcellSizeInPixels,
        XtCCellSizeInPixels,
        XtRInt,
        sizeof(int),
        XtOffset(AppDataPtr, cell_size_in_pixels),
        XtRImmediate,
        (XtPointer) 30,
    },
    {
        XtNdebug,
        XtCDebug,
        XtRBoolean,
        sizeof(Boolean),
        XtOffset(AppDataPtr, debug),
        XtRImmediate,
        (XtPointer) FALSE,
    },
};
```

A resource list entry has the following fields:

- The first two fields of each entry are the name and class of the resource; both strings. These are specified as symbolic constants to improve compile-time checking, and should be selected from *<X11/StringDefs.h>* if any there have an appropriate name, or they can be defined in your application's own include file. For the debug resource entry, the resource name and class strings would be "debug" and "Debug," respectively defined in the application's include file as `XtNdebug` and `XtCDebug`.
- The third field is the representation type. A representation type is a symbolic constant, beginning with `XtR`, that defines the data type of a resource. Because user resource specifications are always in the form of strings while the actual resources may be of any type, Xt uses resource converters to convert the string representation to the actual target type. If necessary, you can define your own representation type in your application include file, but you will also have to provide Xt with a way to convert from string to this type with a type converter function, as described in Chapter 9, *Resource Management and Type Conversion*. Table 3-1 lists the correspondence between the representation types defined in *<X11/StringDefs.h>* and actual C data types or X data types and structures.
- The fourth field, the size of the representation, is specified using the C macro `sizeof` with the actual C type as an argument. For example, when the representation type is `XtRBoolean`, the size field is `sizeof(Boolean)`.
- The fifth field identifies the place in your application's data structure where Xt is to place the converted value. In Example 3-12, the structure is called `AppData`. This field for the debug resource entry is specified using the `XtOffsetOf` macro as `XtOffsetOf(AppData, debug)`. Note that the field name in the application data structure is often the same as the resource name, except with all word transitions marked with an underscore instead of a capital letter.
- The sixth field specifies the representation type of the default value (the seventh field). Xt has routines for converting data between types. They are used to translate strings from the resource databases into certain representation types. Therefore, you can specify the

default value as a string (among other things) and if the default value is needed Xt will convert it to the representation type you specified in field three. If the representation type in field three is not standard in Xt, you will have to write the conversion routine that converts from `XtRString` to that representation, as described in Chapter 9, *Resource Management and Type Conversion*. In Release 3 and in later releases, you can also use the special constant `XtRImmediate` here, which indicates that the value in field seven can be used without conversion. Using `XtRImmediate` should lead to faster startup.

- The seventh field is the default value, using the representation type specified in field six. This default value will be set into the application data structure field if there is no setting for this resource in the resource database.

Table 3-1. Resource Type Strings

Resource Type	Data Type
XtRAcceleratorTable	XtAccelerators
XtRBoolean	Boolean
XtRBool	Bool
XtRCallback	XtCallbackList
XtRColor	XColor
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Display*
XtRFile	FILE*
XtRFloat	float
XtRFont	Font
XtRFontStruct	XFontStruct *
XtRFunction	(*)()
XtRInt	int
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	XtPointer
XtRPosition	Position
XtRShort	short
XtRString	char*
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRWidget	Widget
XtRWindow	Window

3.5.3 Getting the Resources

Once the application data structure and resource list are set up, you pass pointers to them to `XtGetApplicationResources`, just after calling `XtAppInitialize`. `XtGetApplicationResources` will search the databases for any matching resource settings and set the fields in the application data structure.

The last requirement is that you check the values specified by the user to make sure they are acceptable. Example 3-12 shows these two steps from *xbitmap4*.

Example 3-12. Calling XtGetApplicationResources and checking values

```
    .
    .
    .
AppData app_data;
    .
    .
    .
main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget toplevel, vpane, buttonbox, quit, output;
        .
        .
        .
    /* call XtAppInitialize here */

    XtVaGetApplicationResources(toplevel,
            &app_data,
            resources,
            XtNumber(resources),
            /* varargs list here for making
             * application resources
             * non-user-configurable */
            NULL);          /* terminate varargs list */

    /*
     * We must check the application resource values here.
     * Otherwise, user could supply out of range values.
     * Conversion routines do this automatically, so
     * colors are already checked.
     */
    if ((app_data.pixmap_width_in_cells > MAXBITMAPWIDTH) ||
            (app_data.pixmap_width_in_cells < MINBITMAPWIDTH) ||
            (app_data.pixmap_height_in_cells > MAXBITMAPWIDTH) ||
            (app_data.pixmap_height_in_cells < MINBITMAPWIDTH)) {
        fprintf(stderr, "xbitmap: error in resource settings:",
                "bitmap dimension must be between %d and %d cells\n",
                MINBITMAPWIDTH, MAXBITMAPWIDTH);
        exit(1);
    }
    if ((app_data.cell_size_in_pixels < MINCELLSIZE) ||
```

Example 3-12. Calling XtGetApplicationResources and checking values (continued)

```
            (app_data.cell_size_in_pixels > MAXCELLSIZE)) {
        fprintf(stderr, "xbitmap: error in resource settings:",
                "cell size must be between %d and %d pixels\n",
                MINCELLSIZE, MAXCELLSIZE);
        exit(1);
    }
        .
        .
        .
}
```

Only certain of the application resources need their values checked. Because Xt automatically converts the user's color specifications (such as "blue") into the form required by X, it warns the user when a color is specified improperly.* In this case, only the bitmap dimensions and cell size are checked, since they are critical to the application's operation.

3.6 Command-line Options

You already know that Xt automatically customizes widgets according to the resource database, but this is not the whole story. Users often expect command-line arguments for the most important aspects of an application. By default, `XtAppInitialize` understands only a minimal set of command-line arguments; more need to be added so that application resources and certain widget resources can be set from the command line.

There is no point, however, in trying to make every widget resource settable from the command line, because that's the purpose of the resource database. You should concentrate on the resources that the user is most likely to want to have different between two simultaneous instances of the application (since it is difficult to arrange this with the resource database).

Before describing how to define your own command-line arguments, we need to describe what command-line arguments `XtAppInitialize` already handles.

3.6.1 Standard Command-line Options

First of all, `XtAppInitialize` recognizes the *-xrm* option for setting any widget or application resource. For example:

```
spike% xhello -xrm '*background: blue'
```

This option is a little awkward. Not only is it long, but *csh* users must quote the string with right-handed single quotes so that the * is not interpreted by the shell.

*That is, if the resource in a database file is specified properly, but the value is not, Xt will warn the user. However, if the resource is not specified properly, Xt has no way of knowing that the resource was intended to specify a color, and therefore, no message will be issued. For example, "*background: grein" will elicit a warning message because green is misspelled, but "*backgruond: green" will not, because the resource identifier is misspelled.

`XtAppInitialize` also understands some command-line options that were considered so basic that they should be the same for all applications. The above *-xrm* command line can be replaced by:

```
spike% xhello -background blue
```

or:

```
spike% xhello -bg blue
```

`XtAppInitialize` also understands any unique abbreviation of an option name, such as:

```
spike% xhello -backg blue
```

These resources will work with any application written using the X Toolkit—with any widget set. Try the command line above if you have a color screen. If not, try specifying a different font, using the *-fn* or *-font* option.

Table 3-2 lists the complete set of standard options.* Resources starting with a dot rather than an asterisk indicate that that option affects only the application's top-level Shell.

Table 3-2. Standard Command-line Parameters

Option	Resource	Value	Sets
-bg	`*background`	next argument	background color
-background	`*background`	next argument	background color
-bd	`*borderColor`	next argument	border color
-bw	`.borderWidth`	next argument	width of border in pixels
-borderwidth	`.borderWidth`	next argument	width of border in pixels
-bordercolor	`*borderColor`	next argument	color of border
-display	`.display`	next argument	server to use
-fg	`*foreground`	next argument	foreground color
-fn	`*font`	next argument	font name
-font	`*font`	next argument	font name
-foreground	`*foreground`	next argument	foreground color
-geometry	`.geometry`	next argument	size and position
-iconic	`.iconic`	"on"	start as an icon
-name	`.name`	next argument	name of application
-reverse	`*reverseVideo`	"on"	reverse video
-rv	`*reverseVideo`	"on"	reverse video
+rv	`*reverseVideo`	"off"	No Reverse Video
-selectionTimeout	`.selectionTimeout`	Null	selection timeout
-synchronous	`*synchronous`	"on"	synchronous debug mode

*There is no legitimate way for the application to access this parse table, which is necessary to show the user an error in a parameter. Furthermore, this table is in the source for Xt and therefore not available online to some Toolkit users. However, this parse table is not expected to change radically in new releases. Therefore it can probably be safely copied from this document into application code to provide a synopsis of valid options. Most current applications simply say "This application understands all standard X Toolkit command-line options."

Table 3-2. Standard Command-line Parameters (continued)

Option	Resource	Value	Sets
+synchronous	`*synchronous`	"off"	synchronous debug mode
-title	`.title`	next argument	title of application
-xrm	value of argument	next argument	depends on argument

Note that many of these command-line options set the resource of every widget in the application to the same value. A few of them set the resources only of the application's top-level Shell widget.

3.6.2 Defining Your Own Command-line Options

Supplying your own command-line options allows you to simplify the customization of your application. It also allows the user to customize things that are difficult with the resource database (such as setting different values for the same resource in two instances of the application).

You make `XtAppInitialize` understand additional command-line options by initializing a `XrmOptionDescRec` structure (called the *options table*) and passing it as an argument to `XtAppInitialize`. Example 3-13 shows the code that implements command-line options for the application resources added in Examples 3-9, 3-10, and 3-11.

Example 3-13. xbitmap: specifying command-line options

```
/* include files, etc. */
        .
        .
        .
static XrmOptionDescRec options[] = {
    {"-pw",           "*pixmapWidthInCells",  XrmoptionSepArg, NULL},
    {"-pixmapwidth",  "*pixmapWidthInCells",  XrmoptionSepArg, NULL},
    {"-ph",           "*pixmapHeightInCells", XrmoptionSepArg, NULL},
    {"-pixmapheight", "*pixmapHeightInCells", XrmoptionSepArg, NULL},
    {"-cellsize",     "*cellSizeInPixels",    XrmoptionSepArg, NULL},
    {"-fg",           "*foreground",          XrmoptionSepArg, NULL},
    {"-foreground",   "*foreground",          XrmoptionSepArg, NULL},
    {"-debug",        "*debug",               XrmoptionNoArg, "True"},
};

static void Syntax(argc, argv)
int argc;
char * argv[];
{
    int i;
    static int errs = False;

    /* first argument is program name - skip that */
    for (i = 1; i < argc; i++) {
        if (!errs++) /* do first time through */
```

Example 3-13. xbitmap: specifying command-line options (continued)

```
            fprintf(stderr, "xbitmap4: command line option unknown:\n");
            fprintf(stderr, "option: %s\n", argv[i]);
    }

    fprintf(stderr, "xbitmap understands all standard Xt\
            command-line options.\n");

    fprintf(stderr,"Additional options are as follows:\n");
    fprintf(stderr,"Option          Valid Range\n");
    fprintf(stderr,"-pw             MINBITMAPWIDTH to MAXBITMAPWIDTH\n");
    fprintf(stderr,"-pixmapwidth    MINBITMAPWIDTH to MAXBITMAPWIDTH\n");
    fprintf(stderr,"-ph             MINBITMAPHEIGHT to MAXBITMAPHEIGHT\n");
    fprintf(stderr,"-pixmapheight   MINBITMAPHEIGHT to MAXBITMAPHEIGHT\n");
    fprintf(stderr,"-cellsize       MINCELLSIZE to MAXCELLSIZE\n");
    fprintf(stderr,"-fg             color name\n");
    fprintf(stderr,"-foreground     color name\n");
    fprintf(stderr,"-debug          no value necessary\n");
}

main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget toplevel, vpane, buttonbox, quit, output;
      .
      .
      .

    toplevel = XtVaAppInitialize(&app_context;
            "XBitmap4",
            options,              /* command line option table */
            XtNumber(options),
            &argc,
            argv,
            NULL,
            NULL);

    /* XtVaAppInitialize always leaves at least prog name in args */
    if (argc > 1)
        Syntax(argc, argv);
        .
        .
        .
}
```

Each options table entry consists of four fields:

- The option to be searched for on the command line. As with standard command-line options, Xt will automatically accept any unique abbreviation of the option specified here. For example, the option *-pixmapWidthInPixels* will be recognized if typed on the command line as *-pixmapW*. However, if you wanted the option *-pw* to set the same resource, then you would need another entry, since `pw` is not the leading string of `pixmapWidthInPixels`.
- The resource specification. This must identify a widget resource or an application resource, but not provide a value. Since it has the same form as allowed in the resource

databases, it may apply to a single widget or to many widgets. If it applies to no widgets, no error message will be issued.

- The argument style. This field is one of seven constants describing how the option is to be interpreted. These constants are described below in Table 3-3.
- The value. This field is the value to be used for the resource if the argument style is `XrmOptionNoArg`. This field is not used otherwise. Note that this value must already be converted to the value expected for the resource (often not a string). You may be able to use Xt's type converter routines explicitly to convert this data to the right type (see Section 9.3.5).

The `enum` constants that specify the various command-line argument styles are as shown in Table 3-3.

Table 3-3. XrmOptionKind: Command-line Option Style Constants

Constant	Meaning
`XrmoptionNoArg`	Take the value in the `value` field of the options table. For example, this is used for Boolean fields, where the option might be *-debug* and the default value `FALSE`.
`XrmoptionIsArg`	The flag itself is the value without any additional information. For example, if the option were *-on*, the value would be "on." This constant is infrequently used, because the desired value such as "on" is usually not descriptive enough when used as an option (*-on*).
`XrmoptionStickyArg`	The value is the characters immediately following the option with no white space intervening. This is the style of arguments for some UNIX utilities such as *uucico* where *-sventure* means to call system *venture*.
`XrmoptionSepArg`	The next item after the white space after this flag is the value. For example, *-fg blue* would indicate that "blue" is the value for the resource specified by *-fg*.
`XrmoptionResArg`	The resource name and its value are the next argument in `argv` after the white space after this flag. For example, the flag might be: `-res basecalc*background:white;` then the resource name/value pair would be used as is. This form is rarely used because it is equivalent to *-xrm*, and because the C shell requires that special characters such as * be quoted.
`XrmoptionSkipNArgs`	Ignore this option and the next N arguments in `argv`, where N is the value in the last field of this option table entry.
`XrmoptionSkipArg`	Ignore this option and the next argument in `argv`.
`XrmoptionSkipLine`	Ignore this option and the rest of `argv`.

The options table is passed to `XtAppInitialize` as its third argument, and the number of options table entries as the fourth. The `XtNumber` macro is a convenient way to count the number of entries (this is only one of many contexts in which you'll see this macro used).

Note that you *cannot* override the standard options by providing options with the same names in your own parsing table. If you try this, your options with the same names will simply not be set to the values specified on the command line. Instead, the standard options will be set to these values. This was a design decision in Xt, one of the few cases where a user-interface policy is enforced. Uniformity in this basic area was deemed more valuable than flexibility.

Also note that there is no way to instruct Xt to interpret more than one argument format for the same option. For example, you cannot arrange for *-size 3* and *-size3* both to work.

It is important to check whether there is more than one argument left after `XtAppInitialize` has returned. `XtAppInitialize` conveniently removes from `argc` and `argv` the command-line arguments that it has already used to set resources. Command-line options that `XtAppInitialize` doesn't recognize will be left in `argv` and `argc`. This is your chance to catch this error and tell the user.* The `Syntax` function shown in Example 3-12 demonstrates code that informs the user of the proper syntax and the option that was not understood. (In response to incorrect command-line options, UNIX programs traditionally print only the correct calling sequence. However, you can be even nicer to the user by printing the option or options that were in error, by passing `argv` and `argc` into your `Syntax` function, as is done in Example 3-12.

Experienced UNIX programmers will note that Xt applications can (but usually don't) use the single-letter command-line arguments mandated by POSIX and the System V Interface Definition. As mentioned earlier, Xt automatically matches any unique abbreviation for any command-line option. For example, by default the *-display* option can be specified as *-d*, but only if you haven't included any other option in the options table that also begins with d. You can define the meaning of single-letter options simply by including them verbatim in the options table. In other words, if you specify that *-d* turns on a debugging resource, Xt will no longer try to match any other, longer option that also begins with *d*.

Note that the `argc` and `argv` arguments of `XtAppInitialize` are in the same order as in the call to `main`. This is the opposite order of arrays and array lengths throughout other Xt and Xlib routine calls. Also note that the address of `argc`, not `argc` itself, is passed to `XtAppInitialize`. This is so that `XtAppInitialize` can decrement the count to reflect recognized options. Watch out for these snags.

*You can, of course, intentionally treat the arguments remaining after `XtAppInitialize` as filenames or other pieces of data.

3.7 Preventing User Customization of Widget Resources

Although user customization is good to a certain degree, some resources need to be hard-coded to make sure that an application will run properly regardless of user configuration. This is particularly true of widget geometry resources and constraints. For example, it is easy to make a mistake with Form widget constraints such that some important widget is hidden behind others.

You prevent user customization of particular resources by setting them in the calls to `XtVaCreateManagedWidget` or `XtCreateManagedWidget`, using a varargs list or an argument list as was shown in `XtVaSetValues` and `XtSetValues` calls in Section 2.5.1. This section will review those techniques and show a few more tricks.

When resources are set in the calls to create widgets, the application can still modify these resources later using `XtVaSetValues` or `XtSetValues`. Therefore, this is hardcoding only from the user's perspective. Also note that Xt will not tell the user that you have hard-coded certain resources—your application documentation should do this.

The time to hardcode widget resources is when development of an application is almost finished, because until then the resource settings are still in flux and it is easier to edit the application-defaults file than it is to recompile the application.

Selecting which resources to hardcode is something we can't help you with, because there isn't enough experience to go on yet. Some application developers will hardcode almost everything, while others will hardcode nothing. Only time will tell where the proper balance lies.

3.7.1 Using the Varargs Interfaces

Setting widget resources in `XtVaCreateManagedWidget` is the cleanest and most powerful way to do it. You have already seen the style of call shown in Example 3-14, but pay particular attention to the `XtVaTypedArg` entry.

Example 3-14. The R4 varargs interface to creating a widget

```
main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, layout, command;
      .
      .
      .
    layout = XtVaCreateManagedWidget("layout", formWidgetClass,
            topLevel, NULL);

    command = XtVaCreateManagedWidget("command", commandWidgetClass,
            topLevel,          XtNwidth, 200,
            XtVaTypedArg,      XtNbackground,
                XtRString,         "red", strlen("red")+1,
```

Example 3-14. The R4 varargs interface to creating a widget (continued)

```
            XtNjustify,         XtJustifyCenter,
            XtNsensitivity,     TRUE,
            XtNlabel,           "Quit",
            NULL);
    .
    .
    .
}
```

The standard varargs list entry is a resource name/value pair. `XtVaTypedArg` is a special symbol used just before a resource name, which, when encountered, indicates that the next four arguments specify a resource that needs to be converted. In Example 3-14, the value "red" needs to be converted into a pixel value, which is the index to the colormap register that contains the correct RGB values to display on the screen. The argument `XtNbackground` is the resource to be set. `XtRString` is the type of the value to be converted. The final two arguments are the pointer to the value (or the pointer to the pointer to the value if not a string) and the size of the value pointed to in bytes. `XtVaTypedArg` can be used to convert any string that is valid in a resource file but is otherwise difficult to set in the source file. But don't use `XtVaTypedArg` unnecessarily, since it adds overhead.

The varargs interfaces will also accept an arglist as one item in the list, using the symbol `XtVaNestedList` as the resource name. `XtVaNestedList` allows you to create one argument list and pass it to a series of widget creation routines, instead of listing all the same resource settings in each widget creation routine. For more details on this, see Chapter 9, *Resource Management and Type Conversion*.

There is some extra overhead involved in using the varargs interfaces because they massage the arguments into an argument list and call `XtCreateWidget`, `XtCreateManagedWidget`, `XtSetValues`, or `XtGetValues`. However, the added convenience seems worth it. Plus, the `XtVaTypedArg` feature is not supported in the arglist style of call.

Unless you use the `XtVaTypedArg` feature, no diagnostic is printed when you specify a resource value incorrectly in the argument list. Therefore, make sure you specify the resource values correctly, and recognize this as a possible place to begin looking for bugs. (This is another reason to hardcode resources only when the application is almost ready for release. If you do it all at once in a methodical fashion, you are less likely to make mistakes than if you are always adding, subtracting, and changing values in the argument lists.)

NOTE

If you use the varargs style of arguments, but forget to type the Va in the function name, you can get various types of errors. If you specify resources in the call, you will get a core dump at run time, but you can detect the problem with *lint*, which will note the function as having a variable number of arguments. If you don't specify resources in the call, the application will work but you will get the message:

```
Warning: argument count > 0 on NULL argument list
```

3.7.2 Using the Argument List Interfaces

Widget resources can also be hardcoded by creating an argument list containing the resources to be set and their values, and passing it to `XtCreateManagedWidget`.

An *argument list* is just an array of `Arg` structures, of the same type as you set up to call `XtSetValues` or `XtGetValues`. Once set, this array is used as an argument to `XtCreateManagedWidget`. Each `Arg` structure contains a resource/value pair. Attributes specified here override the same ones specified from the command line or from resource databases.

Example 3-15 shows an argument list that hardcodes the sensitivity of a Command widget and its callback list. The sensitivity is a good thing to hardcode for Command widgets, because if set by the user it could disable an application.* The callback list cannot be specified by the user anyway; setting it here is just an alternative to calling `XtAddCallback`.

The `XtArgVal` type used in the callback list aids in porting Toolkit programs to different architectures. It allows the system manufacturer to select the type in the `typedef` for `XtArgVal`, so that the application program need not worry about it. The `value` field of the `Arg` structure may be a number or a pointer.

Example 3-15. An argument list

```
Arg quit_args[] = {
    XtNsensitive,       (XtArgVal) TRUE,
    XtNcallback,        (XtArgVal) quit_callback_list,
};
```

An argument list can be used as an argument in the call to create a widget, as shown in Example 3-16.

Example 3-16. Using an argument list in widget creation

```
/* define quit_args */

main(argc, argv)
int argc;
char *argv[];
{
    Widget quit, box;

    /* create box */
      .
      .
      .

    quit = XtCreateManagedWidget(
            "quit",                 /* widget name */
            commandWidgetClass,     /* widget class */
```

*Sensitivity is a basic resource of all widgets. When set to `TRUE`, the default, a widget accepts input and operates normally. When `FALSE`, a widget displays itself in gray (or otherwise indicates insensitivity) and does not act on user input. The purpose of sensitivity is to allow the application to disable certain commands when they are not valid. If sensitivity were left configurable, the user could turn it off on some widgets and effectively cripple an application. It is hard to imagine the user doing this by accident, but it is not worth taking a gamble.

Example 3-16. Using an argument list in widget creation (continued)

```
            box,                    /* parent widget*/
            quit_args,              /* argument list*/
            XtNumber(quit_args)     /* arglist size */
            );
        .
        .
        .
}
```

Notice the use of the `XtNumber` macro to calculate how many arguments there are in the statically initialized argument list. This macro eliminates the need to keep track of the number of resources you have set.

Note that the `value` field in the argument list must be in the correct representation type for that resource, which is often not a string. You may need to call `XtConvertAndStore` to arrive at the right representation of the data to be placed in the argument list. For details on calling `XtConvertAndStore`, see Chapter 9, *Resource Management and Type Conversion*. An easier way to do this is to use the `XtVaTypedArg` feature supported by the varargs interfaces.

As mentioned earlier, unless you invoke a converter, no diagnostic is printed when you specify a resource value incorrectly in the argument list. Therefore, make sure you specify these correctly.

3.7.2.1 Another Way to Set Arguments

Instead of creating the argument list as a static array, you can allocate storage at run time and use the `XtSetArg` macro to set values into the storage. `XtSetArg` sets a single argument to a resource identifying a constant and a value. Example 3-17 shows the code that would create an argument list with the same contents as the one created in Example 3-15 above. Some people prefer this technique because it places the argument list setting closer to the `XtCreateWidget` call, making the code easier to read. (However, it is still more difficult to read than when using the varargs interfaces.)

Example 3-17. Setting the argument list with XtSetArg

```
int i;
Arg args[10];
/* XtAppInitialize may be called before or after the XtSetArg calls */

i = 0;
XtSetArg(args[i], XtNcallback, (XtArgVal) quit_callback_list); i++;
XtSetArg(args[i], XtNsensitive, (XtArgVal) TRUE); i++;

banner = XtCreateManagedWidget(
        banner,                 /* widget name */
        commandWidgetClass,     /* widget class from Label.h */
        toplevel,               /* parent widget */
        args,                   /* argument list */
        i                       /* arg list size from XtSetArg counter */
);
```

Notice that `i` is used as the number of arguments in the `XtCreateManagedWidget` call, *not* `XtNumber`. `XtNumber` would in this case return the value `10`, the total length of the `Arg` array.

Remember that `i` must be incremented outside the macro, because `XtSetArg` references its first argument twice. The following code will not work because `XtNresource2` will be set into `widget_args[2]` instead of `widget_args[1]`, as desired. Example 3-18 shows you how *not* to use `XtSetArg`:

Example 3-18. Incorrectly setting the argument list with XtSetArg

```
Arg arg[10];
int num_args;

/* This example will NOT work correctly! */

num_args = 0;

/* The next two lines are wrong! */
XtSetArg(args[num_args++], XtNresource1, (XtArgVal) 10);
XtSetArg(args[num_args++], XtNresource2, (XtArgVal) 40);
        .
        .
        .
```

The `XtSetArgs` method has three advantages over static initialization:

- It moves the code that sets the argument list closer to where it is really used in the call to create a widget.
- It allows run-time value modification using the same method.
- The same storage can be used for multiple argument lists.

The disadvantages of the `XtSetArgs` method are as follows:

- It performs all assignments at run time, which slows startup slightly.
- It is possible to try to set more arguments than space has been allocated for, leading to a core dump.
- It is possible to misplace the counter increment, leading to improper operation.
- An argument list and a counter variable must be kept until the call to create the widget instance; the static method requires keeping only the argument list.
- It may waste a small amount of storage since the argument list array is usually initialized larger than the required size.
- Lots of `XtSetArg` calls make for ugly code.

As you can see, these differences are mostly subjective, and you may use whichever method you prefer, or some combination.

3.7.2.2 Merging Argument Lists

`XtMergeArgLists` takes two argument lists and counts all arguments, allocates the storage for a single argument list big enough to hold them, and stores all the entries from both in the returned argument list. It does not remove duplicate entries. The calling application can use `XtNumber` to determine the resulting argument count (or can add the original counts).

3.8 More About Application Contexts

We have used application contexts in the examples so far, but not discussed what they really are or what they are for.

An application context is a structure maintained by Xt that stores all the data associated with the application, including the functions registered by the application and other information about the application. Primarily, its purpose is to allow Xt applications to run without modification on certain operating systems that do not have a separate address space for each process. These systems include the Xerox Cedar environment, the Symbolics Genera environment, and the TI Explorer system. Although systems like this are not common, the goal of all X software in C is to be portable to any system that supports C.*

Why then is the `XtAppContext` exposed in the programming interface? It is possible, though difficult, to create more than one application context within a single program. This is rarely done and its implications are complex, so we will reserve discussion of it until Section 13.6.

The important thing to remember is that, for maximum portability, you need to use the versions of routines that begin with `XtApp` instead of those that don't. For instance, use `XtAppInitialize` instead of `XtInitialize`.

Of the routines you have seen so far in this book, only `XtVaAppInitialize`, `XtAppInitialize`, `XtAppMainLoop`, and `XtAppAddActions` use explicit application contexts. The complete list of routines that have equivalents is presented in Section 13.6.

Throughout this book we will continue to use the routines that use the explicit application context.

*It is almost a maxim that there is no such thing as portable software, only software that has been ported. The goal (and more or less the reality) of X is that the porting process should be much easier than it has traditionally been, and, most important, that only one version of a particular piece of software should need to be maintained. The various idiosyncrasies of particular compilers can be dealt with using conditional preprocessor directives (#ifdef). X features were designed specifically to provide ways to handle differences in screens and keyboards. The application context is an effort to provide a way to handle odd operating systems.

4

An Example Application

This chapter describes a complete application, in several iterations. First, it shows a simple version of the program, a bitmap editor, as it would be written assuming the existence of a BitmapEdit widget (which is actually developed in). Then, two refined versions are developed, each demonstrating additional Toolkit programming techniques. Finally, the same application is shown as it would be written if the bitmap editor were implemented in an application window rather than with the BitmapEdit widget.

In This Chapter:

xbitmap1: Bitmap Editor Using a BitmapEdit Widget 106
Widget Public Functions 109
Application-defaults File 109
xbitmap2: Adding Scrollbars to Handle Large Bitmaps 109
Overriding Translations 114
Action Arguments in the Translation Table 114
The resize_thumbs Action 115
Scrollbar Callbacks 116
xbitmap3: Adding Graphics to Display the Bitmap 118
Graphics from the Application 121
Writing a Bitmap File 124
xbitmap4: Bitmap Editor Without a BitmapEdit Widget 125

4
An Example Application

Enough of these trivial programs! Now for an (almost) real application. This chapter describes the development of a bitmap editor. Although it is simple, it can be easily extended to be quite powerful without any new techniques.

We will show several versions of *xbitmap*, beginning with a simple version that assumes the existence of a BitmapEdit widget (which will actually be developed in Chapter 5, *Inside a Widget*, and Chapter 6, *Basic Widget Methods*). Subsequent versions add scrollbars, then add two widgets that display the bitmap in small scale (one normal, one reverse video), and finally implement the same application without the use of the specialized BitmapEdit widget.

These examples will demonstrate the techniques described in Chapter 2, *Introduction to the X Toolkit*, and Chapter 3, *More Techniques for Using Widgets*, as they would appear in a real application, and will bring up a number of new topics:

- The initial version introduces the bitmap editor application, and demonstrates how the application can access the data within a widget using a public function defined by the widget class.
- The second version, which adds scrollbars, shows how scrollbars are linked to the window that they control.
- The third version demonstrates two ways of adding the small-scale bitmaps, one using a standard widget and one drawing into a Core widget from the application.
- The fourth version does not use the BitmapEdit widget, but instead implements its features by drawing into a Core widget. This demonstrates how to prototype features and prepare for writing your own widget. It also allows us to describe the code necessary to implement the bitmap editor so that you will already understand this code when you see it in the internal framework of a widget described in Chapter 5 and Chapter 6.

Many applications will have at least one custom window for which no existing widget will suffice. This chapter demonstrates that there are several ways to implement such a window, and describes the tradeoffs between the different options. Many features can be added from the application code by creating a Core widget to draw into, or by creating a custom widget. By the end of Chapters 5 and 6, you will have seen how to implement this application entirely with existing widgets and application code, how to implement it using custom widgets, and how to implement a mixture of the two.

4.1 xbitmap1: Bitmap Editor Using a BitmapEdit Widget

The screen appearance of *xbitmap1* is shown in Figure 4-1. As usual, it is a good idea to compile and run each example as it is discussed.*

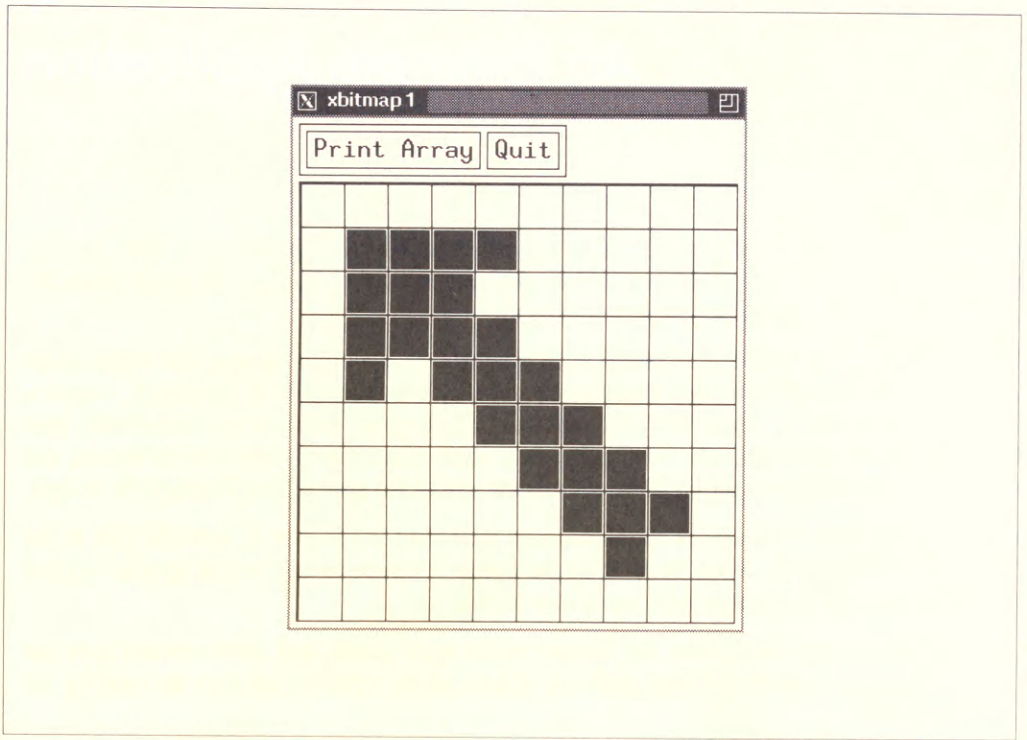


Figure 4-1. xbitmap1: how it looks on the screen

The BitmapEdit widget lets you set bits in the visible bitmap by clicking the first pointer button or dragging the pointer with the first button held down, lets you erase bits using the second button, or lets you toggle bits using the third button. The "Print Output" button simply prints on the standard output an array of 1's and 0's representing the set and unset bits in the bitmap. (Code to read and write standard X11 bitmap files is added in a later version.)

xbitmap1 consists of only five widgets other than the top-level Shell; one Form, one Box, one BitmapEdit, and two Command widgets. The Form widget is the child of the Shell widget. One child of the Form widget is a Box containing the two Command widgets, and the other is the BitmapEdit widget. This arrangement of geometry-managing widgets keeps the appearance of the application neat even when the application is resized.

*How to get and compile the example source code is described in the Preface and Section 2.3.2.

The code for *xbitmap1* is shown in Example 4-1. One new technique shown in this example is the use of the public function `BitmapEditGetArrayString` defined by the BitmapEdit widget. This function allows an application callback function to access the BitmapEdit widget's private data. Even though the array of bits in the bitmap is a resource of the BitmapEdit widget that can be read with `XtGetValues`, this public function provides a more convenient way for the application to get the contents of the bitmap so that it can print them out.

Example 4-1. xbitmap1: complete code

```
 * xbitmap1.c
 */
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Form.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Command.h>

#include "BitmapEdit.h"

Dimension pixmap_width_in_cells, pixmap_height_in_cells;

/*
 * The printout routine prints an array of 1s and 0s representing the
 * contents of the bitmap.  This data can be processed into any
 * desired form, including standard X11 bitmap file format.
 */
/* ARGSUSED */
static void
Printout(widget, client_data, call_data)
Widget widget;
XtPointer client_data;    /* cast to bigBitmap */
XtPointer call_data;      /* unused */
{
    Widget bigBitmap = (Widget) client_data;
    int x, y;
    char *cell;
    cell = BitmapEditGetArrayString(bigBitmap);

    (void) putchar('\n');
    for (y = 0; y < pixmap_height_in_cells; y++) {
        for (x = 0; x < pixmap_width_in_cells; x++)
            (void) putchar(cell[x + y * pixmap_width_in_cells] ? '1' : '0');
        (void) putchar('\n');
    }
    (void) putchar('\n');
}

main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget topLevel, form, buttonbox, quit, output, bigBitmap;

    /* never call a Widget variable "exit"! */
    extern exit();
```

Example 4-1. xbitmap1: complete code (continued)

```
    static XrmOptionDescRec table[] = {
        {"-pw",          "*pixmapWidthInCells",  XrmoptionSepArg, NULL},
        {"-pixmapwidth", "*pixmapWidthInCells",  XrmoptionSepArg, NULL},
        {"-ph",          "*pixmapHeightInCells", XrmoptionSepArg, NULL},
        {"-pixmapheight","*pixmapHeightInCells", XrmoptionSepArg, NULL},
        {"-cellsize",    "*cellSizeInPixels",    XrmoptionSepArg, NULL},
    };

    topLevel = XtVaAppInitialize(
            &app_context,       /* Application context */
            "XBitmap1", /* Application class */
            table, XtNumber(table),   /* command line option list */
            &argc, argv,        /* command line args */
            NULL,               /* for missing app-defaults file */
            NULL);              /* terminate varargs list */

    form = XtVaCreateManagedWidget("form", formWidgetClass,
            topLevel, NULL);

    buttonbox = XtVaCreateManagedWidget("buttonbox", boxWidgetClass,
            form, NULL);

    output = XtVaCreateManagedWidget("output", commandWidgetClass,
            buttonbox, NULL);

    /* callback added below after big bitmap is created */

    quit = XtVaCreateManagedWidget("quit", commandWidgetClass,
            buttonbox, NULL);

    XtAddCallback(quit, XtNcallback, exit, NULL);

    bigBitmap = XtVaCreateManagedWidget("bigBitmap",
            bitmapEditWidgetClass, form, NULL);

    XtAddCallback(output, XtNcallback, Printout, bigBitmap);

    /* need the following values for the printout routine. */
    XtVaGetValues(bigBitmap,
            XtNpixmapWidthInCells, &pixmap_width_in_cells,
            XtNpixmapHeightInCells, &pixmap_height_in_cells,
            NULL);

    XtRealizeWidget(topLevel);

    XtAppMainLoop(app_context);
}
```

You should recognize the command-line options table as the one described in Section 3.6.2. Note that *xbitmap1* does not require application resources, because there is nothing for them to set. The widgets themselves provide all needed resources.

4.1.1 Widget Public Functions

Public functions are supplied by a widget class when it wants certain data to be readable or writable by the application, but for some reason does not want the data both readable *and* writable as would be the case if it were a resource. Sometimes the application needs to be able to read data but not write it. At other times, a widget class has features that it wants controllable from the application but never user-customizable (because they are meaningless at startup, for example), and therefore it provides a function for setting them.

The BitmapEdit widget provides a public function because this version does not allow the application to set the bitmap, and because it is easier for the programmer to get the array with a specialized function than through `XtVaGetValues`. A more complete bitmap editor widget would need to make the array of bits a resource, so that the application could read bitmap files.

4.1.2 Application-defaults File

Example 4-2 shows the application-defaults file for *xbitmap1*. It sets the text in the Command widgets, the size in cells of the bitmap, the size in pixels of each cell in the bitmap, and the foreground color, and constrains the `bigBitmap` widget to appear below the button box. Without this constraint, the Form widget would place the button box overlapping `bigBitmap` instead of above it.

Example 4-2. XBitmap1: application-defaults file

```
*quit*label:    Quit
*output*label:    Print Array
*cellSizeInPixels:    30
*pixmapWidthInCells:    10
*pixmapHeightInCells:    10
*foregroundPixel:    blue
*buttonbox.width:    200
*bigBitmap*fromVert:    buttonbox
```

4.2 xbitmap2: Adding Scrollbars to Handle Large Bitmaps

xbitmap1 always displayed all the cells in the bitmap being edited. This can be inconvenient when the bitmap is large. For example, this makes it difficult to edit any bitmap larger than one-fifth of the screen (because each cell takes up a minimum of five pixels). The solution to this problem is to add scrollbars, and to display only part of the bitmap at a time. We'll add this feature to develop *xbitmap2*, shown in Figure 4-2.

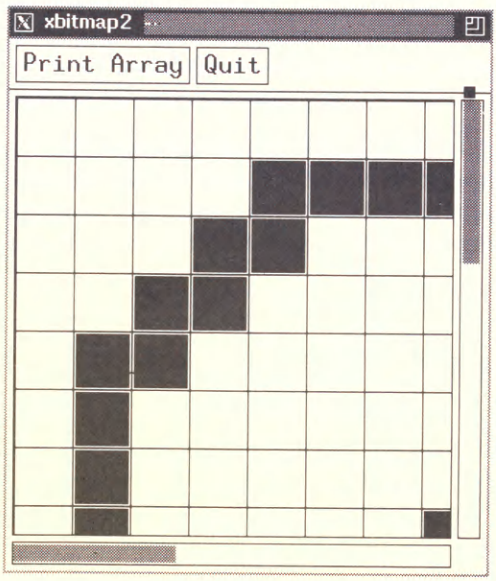


Figure 4-2. xbitmap2: scrollbars added

The easy way to add scrollbars is to use a widget provided just for this purpose, such as the Athena Viewport widget. Simply by inserting a Viewport widget into the instance hierarchy between the Form widget and the BitmapEdit widget and providing the right resource settings, scrollbars will appear around the BitmapEdit widget. This example program is included with the source distribution as *xbitmap5*, but is not described here because it is a simple improvement from *xbitmap1*.

The Viewport widget, like the Dialog widget described in the last chapter, is really just a Form widget, two Scrollbar widgets, and the code to link them together. The window displayed is blank until you create a widget like BitmapEdit as a child of Viewport. This child can be a composite widget with children of its own, if desired.*

*Of course, it is also possible to create a widget similar to Viewport that combines BitmapEdit and scrollbars into a single unit. This widget could be a subclass of a geometry managing widget such as Form, and would create a BitmapEdit widget and the scrollbar widgets as children. It would add the code necessary to create the child widgets and link the scrollbars with BitmapEdit. This widget is described in Chapter 11, *Geometry Management*.

As an exercise, we'll pretend that we don't have the Viewport widget; we'll create scrollbars and link them with the BitmapEdit widget directly from the application. Scrollbars are interesting because they need to be very tightly coupled with the window they are controlling. Whenever the user clicks or drags in the scrollbar, the controlled window must quickly redisplay the correct part of its contents. Each scrollbar widget provides at least two callbacks: one for jump scrolling and one for smooth scrolling. Typically, the first and third buttons initiate jump scrolling by one screenful up and down, respectively, while the second button moves the thumb to the pointer position and smoothly scrolls the window up or down if the pointer is dragged while the button is held down.

The application controls the position and length of the scrollbar thumb. The length of the thumb must reflect the percentage of the data that is currently in view, and the position must reflect the current position in the data. Only the application knows how much data there is. The application must set the thumb as the scrollbar widget is created, and reset the thumb in each of its callbacks. When the scrollbars are resized, this indicates that the accompanying window has also been resized, and therefore the portion of the data being shown has changed. Therefore, the thumb must also be reset in the procedure that handles resizes.

We will use the Athena Scrollbar widget class in this example. Before proceeding further, it might be helpful to read the Scrollbar widget reference page in Volume Five, *X Toolkit Intrinsics Reference Manual*, to get an understanding of the basic operation of the widget. Other widget sets have scrollbar widgets that interact with the application in similar ways.

- Two Scrollbar widgets are created, one for horizontal and one for vertical scrolling.
- Another level of composite widget (a Form widget) is added to manage the BitmapEdit widget and Scrollbar widgets as a unit. The instance hierarchy of *xbitmap2* is shown in Figure 4-3.
- Two callback functions are added for each scrollbar to perform the two kinds of scrolling (smooth and jump scrolling).
- An action is created to reset the scrollbar thumbs when the application is resized. This action is registered with Xt as usual, and added to the existing translation table of the Scrollbar widgets using `XtOverrideTranslations`.

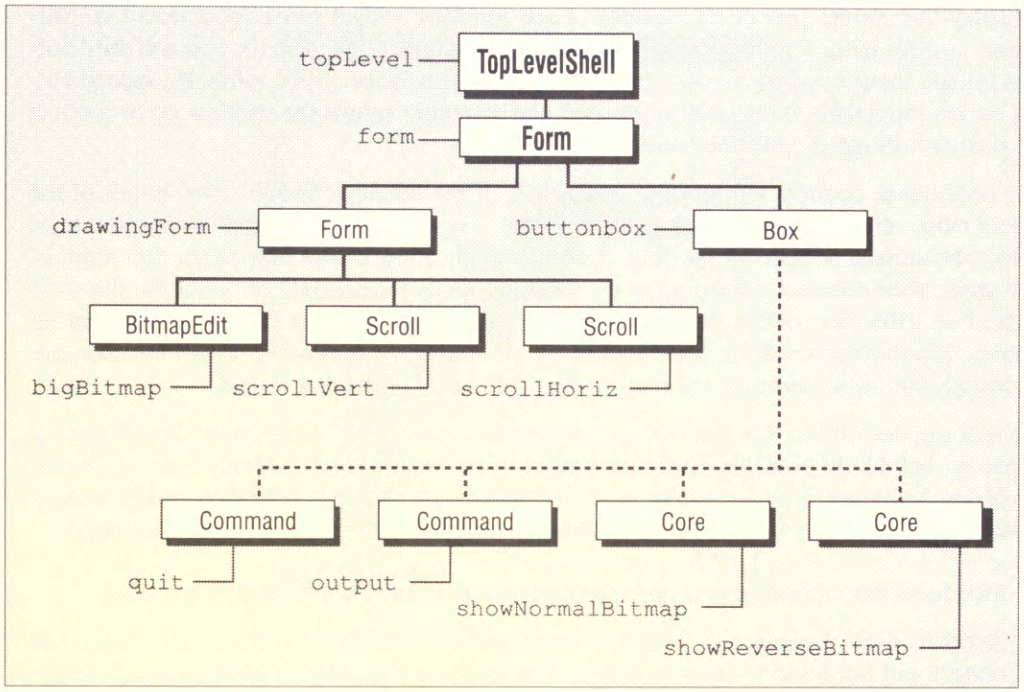


Figure 4-3. The instance hierarchy of xbitmap2

Example 4-3 shows the code needed to add scrollbars to *xbitmap1*.

Example 4-3. xbitmap2: adding scrollbars

```
create_viewport(parent, app_context)
Widget parent;
XtAppContext app_context;
{
    Widget scrollHoriz, scrollVert, drawingForm;
    int cell_size;
    static XtActionsRec window_actions[] = {
        {"resize_thumbs", resize_thumbs}
    };

    XtAppAddActions(app_context, window_actions, 1);

    drawingForm = XtCreateManagedWidget("drawingForm", formWidgetClass,
            parent, NULL, 0);

    bigBitmap = XtCreateManagedWidget("bigBitmap", bitmapEditWidgetClass,
            drawingForm, NULL, 0);

    XtVaGetValues(bigBitmap,
            XtNpixmapHeightInCells, &pixmap_height_in_cells,
            XtNpixmapWidthInCells, &pixmap_width_in_cells,
```

Example 4-3. xbitmap2: adding scrollbars (continued)

```
            XtNcellSizeInPixels, &cell_size,
            NULL);

    pixmap_height_in_pixels = pixmap_height_in_cells * cell_size;
    pixmap_width_in_pixels = pixmap_width_in_cells * cell_size;

    window_width = ((pixmap_width_in_pixels > MAXSIZE) ? MAXSIZE :
            pixmap_width_in_pixels);
    window_height = ((pixmap_height_in_pixels > MAXSIZE) ? MAXSIZE :
            pixmap_height_in_pixels);

    XtVaSetValues(bigBitmap,
            XtNwidth, window_width,
            XtNheight, window_height,
            NULL);

    scrollVert = XtVaCreateManagedWidget("scrollVert",
            scrollbarWidgetClass, drawingForm,
            XtNorientation, XtorientVertical,
            XtNlength, window_height,
            NULL);

    XtAddCallback(scrollVert, XtNscrollProc, scroll_up_down,
            bigBitmap);
    XtAddCallback(scrollVert, XtNjumpProc, jump_up_down, bigBitmap);

    XtOverrideTranslations(scrollVert,
            XtParseTranslationTable("<Expose>: resize_thumbs(v)"));

    scrollHoriz = XtVaCreateManagedWidget("scrollHoriz",
            scrollbarWidgetClass, drawingForm,
            XtNorientation, XtorientHorizontal,
            XtNlength, window_width,
            NULL);

    XtAddCallback(scrollHoriz, XtNscrollProc, scroll_left_right,
            bigBitmap);
    XtAddCallback(scrollHoriz, XtNjumpProc, jump_left_right,
            bigBitmap);
    XtOverrideTranslations(scrollHoriz,
            XtParseTranslationTable("<Expose>: resize_thumbs(h)"));
}
```

The `create_viewport` routine is called from `main` just before calling `XtRealizeWidget` and `XtAppMainLoop`. Think of it as a substitute for the call that created the BitmapEdit widget alone. It is passed the `parent` widget, which, as in *xbitmap1*, is a Form widget whose other child is a Box containing Command widgets. `create_viewport` creates another Form widget to manage the Scrollbar widgets and the BitmapEdit widget. The routine next gets the pixmap dimensions and cell size from the BitmapEdit widget; these three dimensions will be needed in several places in the application, such as in the code that makes the scrollbars the same size as the BitmapEdit widget.

Next, `create_viewport` creates an argument list for each call to create a scrollbar. The arguments set the length and width of each scrollbar, its orientation, and the length of the thumb (the `XtNshown` resource). After creating each widget, the application's two callback functions are registered for that widget. Notice that the ID of the BitmapEdit widget is

passed to the callback functions. The callback functions will use this information to get resource values from the BitmapEdit widget.

The `resize_thumbs` action (shown in Section 4.2.3) simply resets the length of the thumb to represent the current amount of the data being shown, using a public function of the Scrollbar widget. It is invoked whenever the scrollbars are resized, because when the scrollbars change size that means the viewing window also changed size and now displays a different portion of the bitmap. The event that indicates that a widget has been resized is the `ConfigureNotify` event, indicated in the translation table as `Configure`.

4.2.1 Overriding Translations

Finally, `create_viewport` calls `XtOverrideTranslations` to make Xt call the application function `resize_thumbs` in response to resize and expose events. The `resize_thumbs` routine sets the thumb to the proper length (and as a side effect, the Scrollbar widget redraws the thumb). This brings up a number of questions. Why call `XtOverrideTranslations` instead of `XtSetValues`? Also, this translation table includes action functions with arguments. How do these work?

Setting the `XtNtranslations` resource of a widget *replaces* the existing translations of that widget. The problem in this case is that there is no reason to reset translations that we don't want to change because the Scrollbar widgets need them to function. Furthermore, we want the user to be able to configure certain translations, but not the ones that are essential for the application to function. That's why Xt provides two functions for adding translations to an existing table: `XtOverrideTranslations` and `XtAugmentTranslations`. Actions do not work like callback lists—there can be only one action mapped to a particular event or event sequence. The difference between `XtOverrideTranslations` and `XtAugmentTranslations` is in what happens when both the new and existing translation tables have an entry for the same event sequence, but a different action. `XtAugmentTranslations` uses the old translation, whereas `XtOverrideTranslations` uses the new translation.

You should be careful when using `XtOverrideTranslations` not to override any translations that a widget depends on to function. In this case, the application overrides the translation that invokes the Scrollbar widget's expose and resize code, but this is acceptable because Scrollbar does not depend on this code. The Scrollbar widget automatically redraws its contents, the thumb, in response to the Scrollbar widget's public function `XawScrollbarSetThumb`. As long as the application takes over the responsibility to call this function at the right times, the Scrollbar widgets will be properly updated.

4.2.2 Action Arguments in the Translation Table

Notice that the translation tables specified in the calls to `XtOverrideTranslations` in Example 4-4 supply either a `v` or an `h` argument to the action routine `resize_thumbs`. Xt passes this argument as the `call_data` argument to the action routine. This trick simply allows one action routine to serve both the horizontal and vertical scrollbars. This code could also have been written with two independent action routines. It is better to have

independent action routines when the applicable translations are user-configurable (which they are not, in this case). Remember that since the translation table is a string, the argument placed in the translation table is passed to the action routine as a string. This argument, therefore, is basically a constant hardcoded into each translation table. It cannot be used in the same way that the *client_data* argument of callback functions can be used, to pass general data into an action routine.

The action argument is not limited to a single string. You can place a series of words separated by spaces between the parentheses after an action in the translation table. These words will be passed into the action in the third argument, *params*, which is an array of strings just like *argv*. The fourth argument is the number of parameters, *num_params*, analogous to *argc*. (As mentioned earlier, for ANSI C Conformance it is important to declare the *params* and *num_params* arguments even if they are not used in the routine.) Also remember that translation tables must be compiled with `XtParseTranslation-Table` before passing them to `XtOverrideTranslations`, `XtAugment-Translations`, or in an argument list passed to `XtSetValues`.

4.2.3 The resize_thumbs Action

Now, let's move on to the routine referenced in this translation table, `resize_thumbs`. Example 4-4 shows this code. It simply sets the thumb of either scrollbar based on the current position of the data and the size of the BitmapEdit widget's window.

Example 4-4. xbitmap2: resize_thumbs routine

```
/*ARGSUSED*/
static void
Resize_thumbs(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    String *orientation = (String *) params;
    Dimension width, height;
    int cur_x, cur_y;

    XtVaGetValues(bigBitmap,
            XtNheight, &height,
            XtNwidth, &width,
            XtNcurX, &cur_x,
            XtNcurY, &cur_y,
            NULL);

    if (*orientation[0] == 'h')
        XawScrollbarSetThumb(w,
                (float)cur_x/pixmap_width_in_pixels,
                (float)width/pixmap_width_in_pixels);
    else
        XawScrollbarSetThumb(w,
                (float)cur_y/pixmap_height_in_pixels,
                (float)height/pixmap_height_in_pixels);
}
```

The `XawScrollbarSetThumb` routine is defined by the Scrollbar widget class.* It provides an easy method for applications to set the position and length of the thumb. The Scrollbar widget also has resources that set these same parameters.

Note that `resize_thumbs` is called in response to `Expose` events as well as during resizing. This is because the application never knows exactly how much screen space it will be allocated by the window manager. Its size when it appears on the screen may not be the size specified in the code or in the resource databases since the user may intervene to resize the application between two existing windows. Therefore, it is futile to set the thumb when the Scrollbar widgets are created, because the size of the thumbs has to be recalculated based on the current size of the window.

4.2.4 Scrollbar Callbacks

The callback functions for each scrollbar are very similar to `resize_thumbs`. They also reset the thumb, but add code that moves the thumb in proportion to the pointer position. Example 4-5 shows the callback functions for the vertical scrollbar. The horizontal scrollbar has two analogous callback functions.

Example 4-5. xbitmap2: scrolling callback functions

```
/* ARGSUSED */
static void
Scroll_up_down(w, client_data, call_data)
Widget w;
XtPointer client_data;
XtPointer call_data;
{
    int pixels = (int) call_data;
    Dimension height;
    int cur_y;

    XtVaGetValues(bigBitmap,
            XtNheight, &height,
            XtNcurY, &cur_y,
            NULL);

    /* When pixels is negative, right button pressed, move data down,
      * thumb moves up.  Otherwise, left button pressed, pixels
      * positive, move data up, thumb down.
      */
    cur_y += pixels;

    /* limit panning to size of bitmap */
    if (cur_y < 0)
```

*In Release 3 and earlier, all public functions declared by widgets in the Athena widget set had the *Xt* prefix. This was incorrect. Only routines defined by the Xt library should have the *Xt* prefix, because it is important to distinguish the fact that Xt routines are an X Consortium standard, while the Athena widgets are not. Public functions defined by other widget sets will have their own distinctive prefix. As of Release 4, all Athena widget public functions have the prefix *Xaw*.

Example 4-5. xbitmap2: scrolling callback functions (continued)

```
        cur_y = 0;
    else if (cur_y > pixmap_height_in_pixels - height )
        cur_y = pixmap_height_in_pixels - height;

    XtVaSetValues(bigBitmap,
            XtNcurY, cur_y,
            NULL);

    XawScrollbarSetThumb(w,
            (float)cur_y/pixmap_height_in_pixels,
            (float)height/pixmap_height_in_pixels);
}

/* ARGSUSED */
static void
Jump_up_down(w, client_data, call_data)
Widget w;
XtPointer client_data;
XtPointer call_data;
{
    float percent = *(float*)call_data;
    Dimension height;
    int cur_y;

    XtVaGetValues(bigBitmap,
            XtNheight, &height,
            XtNcurY, &cur_y,
            NULL);

    if ((cur_y = (int)(pixmap_height_in_pixels * percent)) >=
            pixmap_height_in_pixels - height)
        cur_y = pixmap_height_in_pixels - height;

    XtVaSetValues(bigBitmap,
            XtNcurY, cur_y,
            NULL);

    XawScrollbarSetThumb(w,
            (float)cur_y/pixmap_height_in_pixels,
            (float)height/pixmap_height_in_pixels);
}
```

Note that the `scroll_up_down` callback is called with the *`pixels`* argument, which describes the vertical position of the button press from the top of the widget. If the left button was pressed, this number is positive, and for the right button it is negative. This routine moves the thumb by this amount, unless the data being shown is already at one of the extremes. In effect, the thumb and data are moved an amount proportional to the vertical distance between the top of the scrollbar and the point where the button press occurred.

The `thumb_up_down` callback moves the top of the thumb to the position where the button press occurred. Since this callback is called repeatedly if the pointer is dragged while the button is pressed, it lets the user scan up and down through the data.

Notice that both callback functions set the `cur_x` and `cur_y` resources of BitmapEdit. This is how the application tells BitmapEdit which part of the bitmap to display. You'll see how this works in Section 4.4, where the code to implement the bitmap editor from the application is described.

4.3 xbitmap3: Adding Graphics to Display the Bitmap

In addition to displaying the bitmap in editable form, with one cell for each pixel in the actual bitmap, a bitmap editor should also display an image in true scale that shows what the bitmap really looks like when displayed on the screen. It is customary to present this bitmap in both normal and reverse video, and update it every time the user toggles a cell in the enlarged bitmap. Figure 4-4 shows the appearance of the application with this feature added. There are two ways to implement this behavior. One is to use the Label widget, which can display a pixmap instead of a string. The other is to draw into a Core widget. From the point of view of the user, both would look exactly the same.

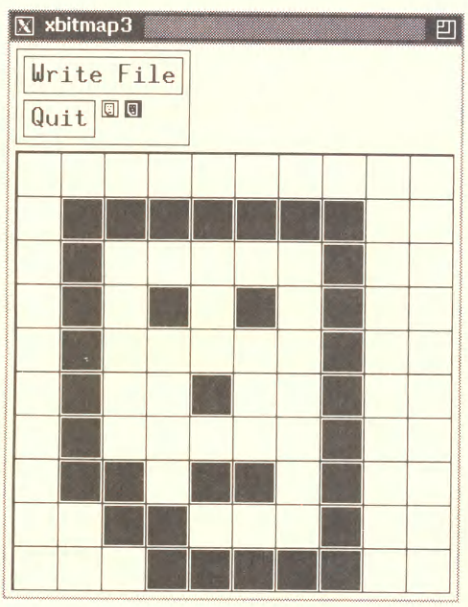


Figure 4-4. xbitmap3: true-scale normal and reverse bitmaps added

Implementing this feature with Label widgets is straightforward; the application creates two pixmaps of the proper size, and sets the `XtNpixmap` resources of two Label widgets to display them. The problem comes when trying to update this display each time a cell is toggled. We can easily draw the changed pixel into the small pixmaps, but without a push the Label widget won't redisplay the pixmap. Because the `XtNpixmap` resource is not changing—the same pixmap ID is used for the life of the application—the Label widget is not notified by Xt that the pixmap has been drawn into. There are two ways to make the existing Label widget redraw the pixmap. One is to clear its window using an Xlib routine that generates an `Expose` event, which will then cause Label to redraw the widget. But this causes

the pixmap to flash annoyingly every time a pixel is changed, since the server clears each window to its background color before the pixmap is redrawn. If you would like to see this problem, or the code to implement this approach, compile and run *xbitmap6*. The second method, which avoids the flashing problem, is to synthesize an `Expose` event (create an `XExposeEvent` structure and fill it with values) and send it to the Label widget using `XSendEvent`. This is a bit of a kludge but it works.

The source of this (surmountable) problem is that the Label widget is designed for static pixmaps. If it provided a public function called `XawLabelRedrawPixmap`, we could more cleanly correct the flashing bug. Although adding this function to the widget code would be easy, we could no longer call the remaining widget class Label. We would be creating a new widget class. (Some widget sets do in fact define two separate widget classes: one for displaying static pixmaps, and the other for dynamic, editable pixmaps.)

An alternate approach is to create Core widgets for each small pixmap and draw into them from the application, as illustrated in Figure 4-5. This way we can redraw each pixmap without clearing the window in between each update, eliminating the flash. We can consider this approach because these small pixmap widgets have no input semantics and simple output semantics and should be easy to implement.

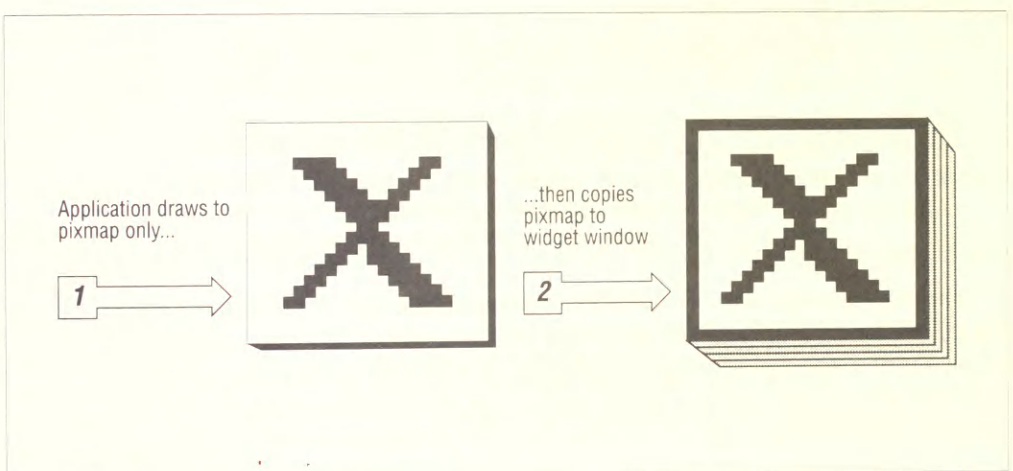


Figure 4-5. Application draws into Pixmap and copies it to widget window

Three new routines are needed: `Redraw_small_picture`, `set_up_things`, and `cell_toggled`. These three routines are primarily composed of Xlib calls that set up for and perform graphics. The `Redraw_small_picture` routine copies a pixmap into a widget, `set_up_things` creates the pixmaps and the GCs (introduced in the next subsection, 4.3.1) needed to draw into these pixmaps, and `cell_toggled` is a callback function registered with the BitmapEdit widget that updates the pixmaps whenever a cell is toggled. Code is also added to `main` to create the two Core widgets and give them translations so that they are redrawn on `Expose` events. We'll show these routines one at a time. Example 4-6 shows the improvements to the `main` routine.

Example 4-6. xbitmap3: implementing small pixmaps by drawing into Core widgets

```
main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget toplevel, form, buttonbox, quit, output;

    static XtActionsRec window_actions[] = {
        {"redraw_small_picture", Redraw_small_picture}
    };

    String trans = "<Expose>:    redraw_small_picture()";
      .
      .
      .
    bigBitmap = XtCreateManagedWidget("bigBitmap",
            bitmapEditWidgetClass, form, NULL, 0);

    XtAddCallback(bigBitmap, XtNcallback, cell_toggled, NULL);

    XtAppAddActions(app_context, window_actions, XtNumber(window_actions));
      .
      .
      .
    XtVaGetValues(bigBitmap,
            XtNpixmapHeightInCells, &pixmap_height_in_cells,
            XtNpixmapWidthInCells, &pixmap_width_in_cells,
            NULL);

    showNormalBitmap = XtVaCreateManagedWidget("showNormalBitmap",
            widgetClass, buttonbox,
            XtNwidth, pixmap_width_in_cells,
            XtNheight, pixmap_height_in_cells,
            XtNtranslations, XtParseTranslationTable(trans),
            NULL);

    showReverseBitmap = XtVaCreateManagedWidget("showReverseBitmap",
            widgetClass, buttonbox,
            XtNwidth, pixmap_width_in_cells,
            XtNheight, pixmap_height_in_cells,
            XtNtranslations, XtParseTranslationTable(trans),
            NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_context);
}
```

Core widgets are created using the class pointer `widgetClass`, rather than `coreWidgetClass` as you might have expected.* The include files for Core (and Composite and Constraint) are included by *<X11/Intrinsics.h>* and therefore don't need to appear in the application.

Note that the size of the Core widget must be set before the Core widgets are realized (either in the application or in the application-defaults file), since the Core widget has a default size of zero and Xt does not allow widgets with a size of zero. Here we use `XtSetArg` to set the

*As of Release 4, `coreWidgetClass` will also work.

size in the argument list used to create the widgets. The same argument list is used to create both widgets.

The `main` routine arranges for the `Redraw_small_picture` routine to be called whenever an `Expose` event arrives for either Core widget, and registers the `cell_toggled` callback with the BitmapEdit widget. These routines depend on the setup performed in `set_up_things`.

4.3.1 Graphics from the Application

Drawing in the X Window System is done by creating a graphics context (GC) that specifies such things as colors and line widths, and then calling a drawing routine that specifies what shape is to be drawn. These two steps are basic to X and required in programs written in any language with or without a toolkit. For example, the call to draw a line specifies only the start and endpoints of the line. The GC specifies everything else about how the server will actually draw this line.

A GC is a server-side object that must be created by the application. The purpose of the GC is to cache on the server side information about how graphics are to be executed by the server, so that this information doesn't have to be sent over the network with every graphics call. If X did not have GCs, every graphics call would have many arguments and this would waste network time (and be annoying to program). Instead, you create a small number of GCs before drawing (in the startup phase of the application). Each represents a particular color or line style you will need to draw with at some point. You then specify one of these GCs in each call to draw. For example, to draw text and be able to highlight it at will, it is customary to create two GCs, one for drawing in white on black, and one for drawing in black on white (where colors can be substituted for black and white on color screens).

Once created, a GC is referred to by its ID, of type `GC`. This ID is specified in calls to draw using that GC.

From the application, the Xlib routine `XCreateGC` is usually used to create GCs. Xt also provides the `XtGetGC` routine for creating GCs, but it is typically used only inside widget code when there could be many of the same GCs created. `XtGetGC` is very similar to `XCreateGC`, except that it arranges for GCs to be shared among widgets (within one application). `XtGetGC` will be described in Section 6.1.

Xt does not provide drawing calls of its own. You must call Xlib directly to draw. An Xlib drawing routine is known as a *primitive*. Under X, text is drawn by a graphics primitive, just as lines, arcs, and other graphics are drawn.

Colors are normally specified by the user as strings such as "blue," but the X server understands colors only when specified as numbers called pixel values. A pixel value is used as an index to a lookup table called a colormap, which contains values for the RGB (red-green-blue) primaries used to generate colors on the screen. However, a particular pixel value does not necessarily always map to the same color, even when run twice on the same system, because the contents of the colormap are configurable on most color systems. The wide variation of graphics hardware that X supports has required that the design of color handling in X be very flexible and very complex.

Fortunately, as long as your widget or application requires only a small number of colors, and you do not particularly care whether the colors you get are exactly the colors you requested, Xt provides a simplified interface. It hides the actual Xlib calls involved in the Resource Manager's converter routines. As described in Chapter 3, *More Techniques for Using Widgets*, if you specify application resources for colors in a resource list, Xt will automatically convert color names specified as resource settings into pixel values. If you need to do more advanced color handling, then you will need to make Xlib calls yourself. For a description of Xlib's color handling, see Chapter 7, *Color* in Volume One, *Xlib Programming Manual*.

All the versions of *xbitmap* so far in this chapter work either in color or monochrome, since the widgets they create can have their color set by resources. (They default to black and white, even on a color screen.) However, to support color in the small pixmaps we are creating takes more work. We would need to add application resources to get the pixel values that will be set into the GCs used in drawing, as described in Section 3.5.

In *xbitmap3*, the `set_up_things` routine is responsible for creating two pixmaps that act as drawing surfaces, and two GCs that will be used to draw and undraw pixels in the pixmaps. Whenever the Core widgets need updating, the pixmaps are copied into the widgets. This is only one possible approach to adding this feature. Another approach is to draw the required points directly into the Core widgets each time they need updating, keeping a record of the drawn points so that the entire widget can be redrawn in case of exposure. `set_up_things` is shown in conjunction with the next version of *xbitmap*, *xbimap4*, at the end of this chapter.

Example 4-7 shows the `cell_toggled` routine. As you may recall, this routine is a callback function registered with the BitmapEdit widget, to be called whenever a cell is toggled.

Example 4-7. xbitmap3: the cell_toggled routine

```
/* ARGSUSED */
static void
Cell_toggled(w, client_data, call_data)
Widget w;
XtPointer client_data;  /* unused */
XtPointer call_data;    /* will be cast to cur_info */
{
    /* cast pointer to needed type: */
    BitmapEditPointInfo *cur_info = (BitmapEditPointInfo *) info;
    /*
      * Note, BitmapEditPointInfo is defined in BitmapEdit.h
      */

    XDrawPoint(XtDisplay(w), normal_bitmap, ((cur_info->mode
            == UNDRAWN) ? draw_gc : undraw_gc), cur_info->newx,
            cur_info->newy);
    XDrawPoint(XtDisplay(w), reverse_bitmap, ((cur_info->mode
            == UNDRAWN) ? undraw_gc : draw_gc), cur_info->newx,
            cur_info->newy);

    /* directly call added actions */
    Redraw_small_picture(showNormalBitmap);
    Redraw_small_picture(showReverseBitmap);
}
```

Note that BitmapEdit passes a structure called `BitmapEditPointInfo` into the callback function as an argument. This structure is defined in the public include file, *BitmapEdit.h*, and it provides the information necessary to keep the small bitmaps displaying the same pattern as BitmapEdit. The fields of `BitmapEditPointInfo` are the mode (whether drawn or undrawn) and the coordinates of the point toggled. The `cell_toggled` routine draws points into the pixmaps according to the information passed in, and then calls `Redraw_small_picture` to copy the pixmaps into each Core widget.

The first line of `cell_toggled` casts the generic pointer `info` into the structure type defined by BitmapEdit, `BitmapEditPointInfo`. This can also be done (perhaps more clearly) by declaring the info argument as type `BitmapEditPointInfo` in the first place. However, some programmers have a preference for using a cast.

The `Redraw_small_picture` routine is shown in Example 4-8.

Example 4-8. xbitmap3: the Redraw_small_picture routine

```
/*ARGSUSED*/
static void
Redraw_small_picture(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    Pixmap pixmap;

    if (w == showNormalBitmap)
        pixmap = normal_bitmap;
    else
        pixmap = reverse_bitmap;

    /*
     * Note that DefaultGCOfScreen is one plane on monochrome
     * screens, but multiple planes on color screens.  The GCs
     * created in set_up_things are always single plane.
     */
    if (DefaultDepthOfScreen(XtScreen(w)) == 1)
        XCopyArea(XtDisplay(w), pixmap, XtWindow(w),
                DefaultGCOfScreen(XtScreen(w)), 0, 0,
                pixmap_width_in_cells, pixmap_height_in_cells, 0, 0);
    else
        XCopyPlane(XtDisplay(w), pixmap, XtWindow(w),
                DefaultGCOfScreen(XtScreen(w)), 0, 0,
                pixmap_width_in_cells, pixmap_height_in_cells, 0, 0, 1);
}
```

This routine is called from `cell_toggled` and by Xt in response to `Expose` events because we registered it as an action and specified it in the translation table resource of each of the Core widgets. The use of one of two Xlib routines, depending on the depth of the screen, is an optimization. `XCopyArea` is faster, but can be used for this job only on monochrome displays, because the pixmaps used here are one plane deep on all displays and must be translated into multiple planes with `XCopyPlane` on color displays.

See Volume One, *Xlib Programming Manual*, for details.

4.3.2 Writing a Bitmap File

Once we have pixmaps in our application that contain the current bitmap, it is a trivial matter to change the `printout` callback function to write a bitmap file instead of just printing an array of 1's and 0's to the standard output. This is easy because there is an Xlib function, `XWriteBitmapFile`, that writes the contents of a single-plane pixmap into a file. Example 4-9 shows the code that gets a filename from the command line and then writes the bitmap file.

Example 4-9. xbitmap3: writing a bitmap file

```
String filename;                    /* filename to read and write */

/* ARGSUSED */
static void
Printout(widget, client_data, call_data)
Widget widget;
XtPointer client_data, call_data; /* unused */
{
    XWriteBitmapFile(XtDisplay(widget), filename, normal_bitmap,
            pixmap_width_in_cells, pixmap_height_in_cells, 0, 0);
}

main(argc, argv)
int argc;
char *argv[];
{
      .
      .
    /* XtAppInitialize */
      .
      .
    if (argv[1] != NULL)
         filename = argv[1];
    else {
         fprintf(stderr, "xbitmap: must specify filename\
             on command line.\n");
         exit(1);
    }

      .
      .
      .
}
```

Contrast this version of `printout` to the one shown in Example 4-1.

Note that reading a bitmap file is not difficult either, but we will not take the space to describe it. If you are curious, the complete code for *xbitmap8*, which both reads and writes bitmap files, is shown in Appendix E, *The xbitmap Application*.

4.4 xbitmap4: Bitmap Editor Without a BitmapEdit Widget

Many applications have at least one window that has certain characteristics not available in any existing widget class in any widget set. The small bitmaps added in the last section provided a simple example of making a custom window by creating a Core widget and drawing into it from the application.

Until you have experience working with widget code, it may be easier to prototype the "custom window" for your application by adding to a Core widget using just the techniques described so far in this book. Once this code is working, and you have read Chapter 5, *Inside a Widget*, and Chapter 6, *Basic Widget Methods*, you can easily move the code into a widget when you want to package it or take advantage of any of the features of Xt that are inaccessible from the application.

The code for the BitmapEdit widget was originally written in an application, and later moved into a widget. In this section we describe this original application, a version of *xbitmap* that works just like *xbitmap1* but without using the BitmapEdit widget. `xbitmap4` is a culmination of many of the techniques described so far in this book. In addition, you will be seeing this same code inside the BitmapEdit widget in Chapters 5 and 6, and it should be easier to understand the additional widget code when you have already seen the functional code described in a familiar setting.

This example takes advantage of application resources to set the configurable parameters of the bitmap code. The code that sets up the application resources was described in Section 3.5. When moving the code into a widget framework, the same resource list will be used verbatim. The example also provides command-line options to set the important parameters of the bitmap code. The code for processing these options was described in Section 3.6. The code is the same whether used for setting application resources or widget resources, except no call equivalent to `XtGetApplicationResources` is necessary in a widget.

The exposure strategy used for the bitmap editor is the same as for the small bitmaps in the previous section. The application creates a large pixmap of depth one that stores the current image of the bitmap being edited. Whenever the screen needs updating, the applicable part of the pixmap is copied to the Core widget in the `Redraw_picture` routine. Because this pixmap is much bigger than the ones in the last section, it is an important optimization that only the required parts of the pixmap are copied. (This is not the only possible exposure strategy. This particular strategy has very low network load, but uses a relatively large amount of server memory. For this reason it is not ideal for PC servers.)

The `set_up_things` routine creates the pixmap, draws a grid into it that will persist for the life of the application, and creates three GCs. One GC is for copying from the pixmap to the window, and two are for drawing and undrawing cells in the pixmap. The `btn_event` routine draws and undraws cells in the pixmap according to pointer clicks and drags, and calls `Redraw_picture` to update the Core widget display.

`Redraw_picture` is called both from the application and from Xt. This is a common trick used to reduce the duplication of drawing code. Since `Redraw_picture` is an action, it has an event argument that is used by Xt to pass in the `Expose` event describing the area exposed. This application also uses this argument by constructing a fake event to pass in information about which part of the widget to draw.

The application adds actions and sets the `XtNtranslations` resource of the Core widget so that Xt calls the application routine `Redraw_picture` whenever `Expose` events arrive, and calls `btn_event` when `ButtonPress` or `MotionNotify` events arrive.

Example 4-10 shows the complete code for *xbitmap4*. You have seen all the techniques here in various examples before. You should work through the code and make sure you understand the purpose of each section. However, don't worry about the details of the Xlib calls, since they are specific to this application.

Example 4-10. xbitmap4: implementing the bitmap editor from the application

```
/*
 * xbitmap4.c
 */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Paned.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Command.h>

#include <stdio.h>

/*
 * The following could be placed in an "xbitmap.h" file.
 */
#define XtNdebug "debug"
#define XtCDebug "Debug"
#define XtNpixmapWidthInCells "pixmapWidthInCells"
#define XtCPixmapWidthInCells "PixmapWidthInCells"
#define XtNpixmapHeightInCells "pixmapHeightInCells"
#define XtCPixmapHeightInCells "PixmapHeightInCells"
#define XtNcellSizeInPixels "cellSizeInPixels"
#define XtCCellSizeInPixels "CellSizeInPixels"

#define DRAWN 1
#define UNDRAWN 0

#define DRAW 1
#define UNDRAW 0

#define MAXLINES  1000

#define MINBITMAPWIDTH  2
#define MAXBITMAPWIDTH  1000
#define MINBITMAPHEIGHT  2
#define MAXBITMAPHEIGHT  1000
#define MINCELLSIZE  4
#define MAXCELLSIZE  100

#define SCROLLBARWIDTH 15

Pixmap big_picture;
GC draw_gc, undraw_gc; /* for drawing into the big_picture,
                        * 1-bit deep */
GC copy_gc;            /* for copying from pixmap into window,
                        * screen depth */
Widget bitmap;         /* drawing surface */
char *cell;            /* array for printing output and keeping
                        * track of cells drawn */
int cur_x, cur_y;
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
Dimension pixmap_width_in_pixels, pixmap_height_in_pixels;

/* data structure for application resources */
typedef struct {
    Pixel copy_fg;
    Pixel copy_bg;
    int pixmap_width_in_cells;
    int pixmap_height_in_cells;
    int cell_size_in_pixels;
    Boolean debug;
} AppData, *AppDataPtr;

AppData app_data;

/* resource list */
static XtResource resources[] = {
    {
        XtNforeground,
        XtCForeground,
        XtRPixel,
        sizeof(Pixel),
        XtOffsetOf(AppData, copy_fg),
        XtRString,
        XtDefaultForeground
    },
    {
        XtNbackground,
        XtCBackground,
        XtRPixel,
        sizeof(Pixel),
        XtOffsetOf(AppData, copy_bg),
        XtRString,
        XtDefaultBackground
    },
    {
        XtNpixmapWidthInCells,
        XtCPixmapWidthInCells,
        XtRInt,
        sizeof(int),
        XtOffsetOf(AppData, pixmap_width_in_cells),
        XtRImmediate,
        (XtPointer) 32,
    },
    {
        XtNpixmapHeightInCells,
        XtCPixmapHeightInCells,
        XtRInt,
        sizeof(int),
        XtOffsetOf(AppData, pixmap_height_in_cells),
        XtRImmediate,
        (XtPointer) 32,
    },
    {
        XtNcellSizeInPixels,
        XtCCellSizeInPixels,
        XtRInt,
        sizeof(int),
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
        XtOffsetOf(AppData, cell_size_in_pixels),
        XtRImmediate,
        (XtPointer) 30,
    },
    {
        XtNdebug,
        XtCDebug,
        XtRBoolean,
        sizeof(Boolean),
        XtOffsetOf(AppData, debug),
        XtRImmediate,
        (XtPointer) FALSE,
    },
};

/* Command-line options table */
static XrmOptionDescRec options[] = {
    {"-pw",           "*pixmapWidthInCells",    XrmoptionSepArg, NULL},
    {"-pixmapwidth",  "*pixmapWidthInCells",    XrmoptionSepArg, NULL},
    {"-ph",           "*pixmapHeightInCells",   XrmoptionSepArg, NULL},
    {"-pixmapheight", "*pixmapHeightInCells",   XrmoptionSepArg, NULL},
    {"-cellsize",     "*cellSizeInPixels",      XrmoptionSepArg, NULL},
    {"-fg",           "*foreground",            XrmoptionSepArg, NULL},
    {"-foreground",   "*foreground",            XrmoptionSepArg, NULL},
    {"-debug",        "*debug",                 XrmoptionNoArg, "True"},
};

/* callback function to print cell array to stdout */
/* ARGSUSED */
static void
Printout(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    int x, y;
    putchar('\n');
    for (y = 0; y < app_data.pixmap_height_in_cells; y++) {
        for (x = 0; x < app_data.pixmap_width_in_cells; x++)
                putchar(cell[x + y * app_data.pixmap_width_in_cells]
                        ? '1' : '0');
        putchar('\n');
    }
    putchar('\n');

    /*
     * It wouldn't be hard to write a function to convert from cell format
     * to one accepted by XWriteBitmapFile, and the reverse to import
     * such files.  This is done in xbitmap3, where it is even easier.
     */
}

static void RedrawPicture(), DrawCell(), UndrawCell(), ToggleCell(),
       DrawPixmaps();

static void Syntax(argc, argv)
int argc;
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
char * argv[];
{
    int i;
    static int errs = False;

    /* first argument is program name - skip that */
    for (i = 1; i < argc; i++) {
        if (!errs++) /* do first time through */
            fprintf(stderr, "xbitmap4: command line option not
                understood:\n");
        fprintf(stderr, "option: %s\n", argv[i]);
    }

    fprintf(stderr, "xbitmap understands all standard Xt command\
            line options.\n");

    fprintf(stderr, "Additional options are as follows:\n");
    fprintf(stderr, "Option         Valid Range\n");
    fprintf(stderr, "-pw            MINBITMAPWIDTH to MAXBITMAPWIDTH\n");
    fprintf(stderr, "-pixmapwidth   MINBITMAPWIDTH to MAXBITMAPWIDTH\n");
    fprintf(stderr, "-ph            MINBITMAPHEIGHT to MAXBITMAPHEIGHT\n");
    fprintf(stderr, "-pixmapheight MINBITMAPHEIGHT to MAXBITMAPHEIGHT\n");
    fprintf(stderr, "-cellsize      MINCELLSIZE to MAXCELLSIZE\n");
    fprintf(stderr, "-fg            color name\n");
    fprintf(stderr, "-foreground    color name\n");
    fprintf(stderr, "-debug         no value necessary\n");
}

main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget topLevel, vpane, buttonbox, quit, output;
    extern exit();
    /* translation table for bitmap core widget */
    String trans =
            "<Expose>:      RedrawPicture()         \n\
             <Btn1Down>:    DrawCell()              \n\
             <Btn2Down>:    UndrawCell()            \n\
             <Btn3Down>:    ToggleCell()            \n\
             <Btn1Motion>:  DrawCell()              \n\
             <Btn2Motion>:  UndrawCell()            \n\
             <Btn3Motion>:  ToggleCell()";

    static XtActionsRec window_actions[] = {
        {"RedrawPicture",   RedrawPicture},
        {"DrawCell",        DrawCell},
        {"UndrawCell",      UndrawCell},
        {"ToggleCell",      ToggleCell},
    };

    topLevel = XtVaAppInitialize(
            &app_context,       /* Application context */
            "XBitmap4",
            options, XtNumber(options),
            &argc, argv,        /* command line args */
            NULL,               /* for missing app-defaults file */
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
            NULL);                  /* terminate varargs list */

    /* XtInitialize leaves program name in args */
    if (argc > 1)
        Syntax(argc, argv);

    XtGetApplicationResources(topLevel,
            &app_data,
            resources,
            XtNumber(resources),
            NULL,
            0);

    /*
     * We must check the application resource values here.
     * Otherwise, user could supply out of range values and
     * crash program.  Conversion routines do this automatically,
     * so colors are already checked.
     */
    if ((app_data.pixmap_width_in_cells > MAXBITMAPWIDTH) ||
            (app_data.pixmap_width_in_cells < MINBITMAPWIDTH) ||
            (app_data.pixmap_height_in_cells > MAXBITMAPWIDTH) ||
            (app_data.pixmap_height_in_cells < MINBITMAPWIDTH)) {
        fprintf(stderr, "xbitmap: error in resource settings:\
                dimension must be between %d and %d cells\n",
                MINBITMAPWIDTH, MAXBITMAPWIDTH);
        exit(1);
    }
    if ((app_data.cell_size_in_pixels < MINCELLSIZE) ||
            (app_data.cell_size_in_pixels > MAXCELLSIZE)) {
        fprintf(stderr, "xbitmap: error in resource settings:\
                cell size must be between %d and %d pixels\n",
                MINCELLSIZE, MAXCELLSIZE);
        exit(1);
    }

    /* begin application code */

    set_up_things(topLevel);

    cell = XtCalloc(app_data.pixmap_width_in_cells *
            app_data.pixmap_height_in_cells, sizeof(char));

    if (app_data.debug)
        fprintf(stderr, "xbitmap: pixmap dimensions are %d by %d\n",
                app_data.pixmap_width_in_cells,
                app_data.pixmap_height_in_cells);

    vpane = XtVaCreateManagedWidget("vpane", panedWidgetClass, topLevel,
            XtNwidth, pixmap_width_in_pixels,
            NULL);

    buttonbox = XtCreateManagedWidget("buttonbox", boxWidgetClass,
            vpane, NULL, 0);

    output = XtCreateManagedWidget("output", commandWidgetClass,
            buttonbox, NULL, 0);

    XtAddCallback(output, XtNcallback, Printout, NULL);

    quit = XtCreateManagedWidget("quit", commandWidgetClass,
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
            buttonbox, NULL, 0);

    XtAddCallback(quit, XtNcallback, exit, NULL);

    bitmap = XtVaCreateManagedWidget("bitmap", widgetClass, vpane,
            XtNtranslations, XtParseTranslationTable(trans),
            XtNwidth, pixmap_width_in_pixels,
            XtNheight, pixmap_height_in_pixels,
            NULL);

    XtAppAddActions(app_context, window_actions, XtNumber
            (window_actions));

    XtRealizeWidget(topLevel);

    XtAppMainLoop(app_context);
}

set_up_things(w)
Widget w;
{
    XGCValues values;
    int x, y;
    XSegment segment[MAXLINES];
    int n_horiz_segments, n_vert_segments;

    pixmap_width_in_pixels = app_data.pixmap_width_in_cells *
            app_data.cell_size_in_pixels;
    pixmap_height_in_pixels = app_data.pixmap_height_in_cells *
            app_data.cell_size_in_pixels;

    big_picture = XCreatePixmap(XtDisplay(w),
            RootWindowOfScreen(XtScreen(w)),
            pixmap_width_in_pixels, pixmap_height_in_pixels, 1);

    values.foreground = 1;
    values.background = 0;
    values.dashes = 1;
    values.dash_offset = 0;
    values.line_style = LineOnOffDash;

    draw_gc = XCreateGC(XtDisplay(w), big_picture,
            GCForeground | GCBackground | GCDashOffset | GCDashList |
            GCLineStyle, &values);

    values.foreground = 0;
    values.background = 1;
    undraw_gc = XCreateGC(XtDisplay(w), big_picture,
            GCForeground | GCBackground | GCDashOffset | GCDashList |
            GCLineStyle, &values);

    values.foreground = app_data.copy_fg;
    values.background = app_data.copy_bg;
    copy_gc = XCreateGC(XtDisplay(w), RootWindowOfScreen(XtScreen(w)),
            GCForeground | GCBackground, &values);

    XFillRectangle(XtDisplay(w), big_picture, undraw_gc, 0, 0,
            pixmap_width_in_pixels, pixmap_height_in_pixels);

    /* draw permanent grid into pixmap */
    n_horiz_segments = app_data.pixmap_height_in_cells + 1;
    n_vert_segments = app_data.pixmap_width_in_cells + 1;
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
    for (x = 0; x < n_horiz_segments; x += 1) {
        segment[x].x1 = 0;
        segment[x].x2 = pixmap_width_in_pixels;
        segment[x].y1 = app_data.cell_size_in_pixels * x;
        segment[x].y2 = app_data.cell_size_in_pixels * x;
    }

    /* drawn only once into pixmap */
    XDrawSegments(XtDisplay(w), big_picture, draw_gc, segment,
            n_horiz_segments);

    for (y = 0; y < n_vert_segments; y += 1) {
        segment[y].x1 = y * app_data.cell_size_in_pixels;
        segment[y].x2 = y * app_data.cell_size_in_pixels;
        segment[y].y1 = 0;
        segment[y].y2 = pixmap_height_in_pixels;
    }

    /* drawn only once into pixmap */
    XDrawSegments(XtDisplay(w), big_picture, draw_gc, segment,
            n_vert_segments);
}

/* ARGUSED*/
static void
RedrawPicture(w, event, params, num_params)
Widget w;
XExposeEvent *event;
String *params;
Cardinal *num_params
{
    register int x, y;
    unsigned int width, height;

    if (event) {    /* drawing because of expose or button press */
        x = event->x;
        y = event->y;
        width = event->width;
        height =  event->height;
    }
    else {    /* drawing because of scrolling */
        x = 0;
        y = 0;
        width =  10000;  /* always the whole window! */
        height =  10000;
    }

    if (DefaultDepthOfScreen(XtScreen(w)) == 1)
        XCopyArea(XtDisplay(w), big_picture, XtWindow(w),
                copy_gc, x + cur_x,
                y + cur_y, width, height, x, y);
    else
        XCopyPlane(XtDisplay(w), big_picture, XtWindow(w),
                copy_gc, x + cur_x,
                y + cur_y, width, height, x, y, 1);
}

/* ARGUSED*/
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
static void
DrawCell(w, event, params, num_params)
Widget w;
XButtonEvent *event;
{
    DrawPixmaps(draw_gc, DRAW, w, event);
}

/* ARGUSED */
static void
UndrawCell(w, event, params, num_params)
Widget w;
XButtonEvent *event;
String *params;
Cardinal *num_params;
{
    DrawPixmaps(undraw_gc, UNDRAW, w, event);
}

/* ARGUSED */
static void
ToggleCell(w, event)
Widget w;
XButtonEvent *event;
String *params;
Cardinal *num_params;
{
    static int oldx = -1, oldy = -1;
    GC gc;
    int mode;
    int newx = (cur_x + event->x) / app_data.cell_size_in_pixels;
    int newy = (cur_y + event->y) / app_data.cell_size_in_pixels;

    if ((mode = cell[newx + newy * app_data.pixmap_width_in_cells])
            == DRAWN) {
        gc = undraw_gc;
        mode = UNDRAW;
    }
    else {
        gc = draw_gc;
        mode = DRAW;
    }

    if (oldx != newx || oldy != newy) {
        oldx = newx;
        oldy = newy;
        DrawPixmaps(gc, mode, w, event);
    }
}

/* Private Function */
static void
DrawPixmaps(gc, mode, w, event)
GC gc;
int mode;
Widget w;
XButtonEvent *event;
{
```

Example 4-10. xbitmap4: implementing the bitmap editor from the application (continued)

```
    int newx = (cur_x + event->x) / app_data.cell_size_in_pixels;
    int newy = (cur_y + event->y) / app_data.cell_size_in_pixels;
    XExposeEvent fake_event;

    /* if already done, return */
    if (cell[newx + newy * app_data.pixmap_width_in_cells] == mode)
        return;

    XFillRectangle(XtDisplay(w), big_picture, gc,
            app_data.cell_size_in_pixels*newx + 2,
            app_data.cell_size_in_pixels*newy + 2,
            (unsigned int)app_data.cell_size_in_pixels - 3,
            (unsigned int)app_data.cell_size_in_pixels - 3);

    cell[newx + newy * app_data.pixmap_width_in_cells] = mode;

    fake_event.x = app_data.cell_size_in_pixels * newx - cur_x;
    fake_event.y = app_data.cell_size_in_pixels * newy - cur_y;
    fake_event.width = app_data.cell_size_in_pixels;
    fake_event.height = app_data.cell_size_in_pixels;

    RedrawPicture(bitmap, &fake_event);
```

5

Inside a Widget

This chapter describes the code inside a basic widget. Much of this code is common to all widgets. You can think of it as a framework that Xt uses to implement a widget's features. After reading this chapter, you should understand the procedure for creating your own widget around this framework.

In This Chapter:

Widget Source File Organization 138
The Private Header File—BitmapEdiP.h 138
Parts and Records 139
Class Part and Class Record 140
Instance Part and Instance Record 141
The Widget Implementation File—BitmapEdit.c 143
Obligatory Include Files 145
Defining the Resource List 145
The Translation Table and Actions Table 148
Declaring Methods 150
Initializing the Class Record 151
The Core Class Part 151
Initializing the Core Methods 153
Description of Core Methods 155
Packaging the Class Record for Application Use 157
A Sample Method 157
The Public Header File—BitmapEdit.h 160
The Process of Widget Writing 163
Summary of Conventions 164

5
Inside a Widget

This chapter reveals the framework of code common to all widgets. As an example, it describes the code for the BitmapEdit widget that was used in versions of the *xbitmap* applications early in Chapter 4, *An Example Application*. Later examples in that chapter described how to implement the bitmap editor without a BitmapEdit widget. Therefore, you have already seen the code that is specific to this widget, and can concentrate on the framework and how to place widget-specific code within this framework.

Some applications use only the standard user interface elements defined in the widget set. If you are writing an application like this, you have no need to write your own widgets, and no real need to understand the internals of widgets. However, many applications have at least one custom window which has features not supported by any existing widget. To implement such features, you can add to a Core widget or you can write your own widget.

Placing specialized user-interface code into a widget has several advantages. For one, the widget becomes a self-contained module that can be modified and documented separately and used in other programs. Second, the code will take advantage of Xt's automatic dispatching of events to widgets, and several other features that you can't use from the application. Finally, it is a general premise of Xt that application code should be separated as much as possible from user-interface code.

It is important to remember that widget classes are never written from scratch. People always start from template files that contain the framework for a widget without the specific code, or even better, if possible, from an existing widget that has some similar characteristics. Therefore, you'll never have to type in the framework you're about to see. You'll only have to learn where to insert your code into it. The beauty of the widget framework is that the code within it is very modular—each module has a specific place and purpose. Once you understand the framework, you can locate these modules in existing code and use them as examples. Chapter 6, *Basic Widget Methods*, shows you how to write the most important modules within the framework.

5.1 Widget Source File Organization

A widget is implemented in two include files and an executable code file. Each of these files contains specific elements. The names of these files are derived from the name of the widget class, which in this case is BitmapEdit:

- The private header file, *BitmapEdiP.h*, defines the widget's class and instance structures, including pointers to the widget's methods.
- The implementation file, *BitmapEdit.c*, contains the actual code for the widget, including the widget's methods and actions.
- The public header file, *BitmapEdit.h*, contains declarations needed by the application to use the widget.

The final "P" in the include file name stands for Private. Only *BitmapEdit.c* and any modules that implement subclasses of BitmapEdit should include *BitmapEdiP.h*. If an application includes this file and references any of its contents, it is breaking the rules of encapsulation, and changes to this widget might affect the application.

An application program that uses a widget includes only *BitmapEdit.h*.

The implementation filenames (and all the filenames used by your application, for that matter) should have 12 or fewer characters so that the code can be copied easily to some System V systems.* That's why the "t" in *BitmapEdiP.h* is left out.

The next three major sections describe the contents of the three files that make up a widget. These files are treated in the order shown in the above list because they are generally developed in this order.

5.2 The Private Header File—BitmapEdiP.h

Xt implements classes and instances with two structures, the class structure and the instance structure. By definition of C structures, the fields in both structures are fixed at compile time. Both of these structures are defined in the private include file for the class, in this case *BitmapEdiP.h*.

There is only one copy of the class structure in the running application, which all instances share. But each instance has its own copy of the instance structure, whose fields are set by the resource database as the instance is created. Fields in the instance structure can also be set by `XtVaSetValues` or read by `XtVaGetValues`.

*These systems have a 14-character filename limit; the 12-character limit allows files to be placed under source control or compressed. The X Consortium keeps all filenames to a maximum of 12 characters. That limits the actual class name to 9 characters, to leave room for the P.h suffix. If the widget class name is longer than 9 characters, it is truncated in the filename. For example, the private include file for the Constraint class is *ConstrainP.h* (without the final t); *BitmapEdiP.h* is *BitmapEdiP.h* in our example code.

The class structure's fields contain pointers to methods and pieces of data that control how Xt handles instances of this class.

The instance structure carries a complete widget state, including everything that can be different between one instance and the next. For example, the instance structure of the Core class, which every widget inherits, has fields for `x`, `y`, `width`, and `height` values, which correspond to the size of the widget and its location relative to the top-left corner of its parent. These particular fields are public; they can be set from the resource database or with `XtVaSetValues`, and their values can be read with `XtVaGetValues`. Other instance structure fields are private, and are used only for convenience to make data globally available within the widget.

Actually, the instance structure is not global in the normal C sense. When Xt invokes a method, it passes the method a pointer to the widget instance structure. Methods do their work by using the public fields and changing the private fields in this instance structure. For example, a method that draws in the window can get the window's dimensions directly from the widget instance structure. When action functions are called, they also are passed a pointer to the instance structure. Therefore, the fields inside the instance structure are available just about everywhere in the widget code.

5.2.1 Parts and Records

The organization of both class and instance structures is determined by the hierarchy of widget classes from which the current widget class is derived. For example, in the code for the BitmapEdit widget whose class hierarchy is shown in Figure 5-1, the class structure begins with the class fields defined by the Core class, followed by the class fields defined by BitmapEdit.

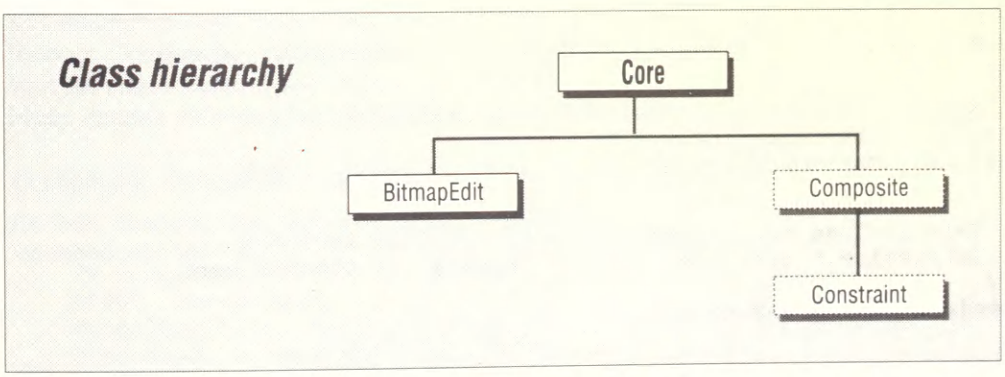


Figure 5-1. The class hierarchy of the BitmapEdit widget (with other classes shown dotted)

Xt supplies three basic classes that are used as the basis for custom widgets, Core, Composite, and Constraint.* Figure 5-1 also shows the relationship of these classes to each other.

*Shell, the fourth Intrinsics-supplied widget class, is not usually subclassed by application or widget programmers.

Core is the class upon which all widgets are based. It defines common characteristics of all widgets, such as their methods, and basic resources such as height and width. Even if your widget is unlike any existing widget, it will still inherit features from the Core widget. The class and instance structures of a subclass of Composite such as Box begin with the fields from Core, continue with fields from Composite, and end with the fields defined by Box. Composite and Constraint are subclasses of Core that have additional methods that allow them to manage children; they are described in Chapter 11, *Geometry Management*. This chapter concentrates on the features of the Core widget.

Xt requires that you implement each class's new fields as a separate structure called a `Part`, and then combine this `Part` structure with each superclass's `Part` structure in the complete structure called a `Rec`, or record. In real code, these structures are called *widgetname*`ClassPart` and *widgetname*`ClassRec` for the class structure, and simply *widgetname*`Part` and *widgetname*`Rec` for the instance structure.

The reason for this "structure within a structure" design is primarily to reduce the changes in the code of subclasses that would be required if changes were made to the superclass's structures. As we will see later, only the portion of the *.c* file that initializes the class structure will need changing if a superclass class structure is changed. A second benefit of this design is that it reduces the amount of typing required to implement a new class.

5.2.2 Class Part and Class Record

Let's make these ideas concrete by showing the class structure for BitmapEdit. The complete class structure is called `BitmapEditClassRec`, and the partial structure is called `BitmapEditClassPart`. Their definitions from *BitmapEdiP.h* are shown in Example 5-1.

Example 5-1. BitmapEdiP.h: the class part and class record

```
/*
 * BitmapEditP.h - Private definitions for BitmapEdit widget
 */

/* protect against multiple including of this file */
#ifndef _ORABitmapEditP_h
#define _ORABitmapEditP_h

/*
 * This include not needed unless the .c file includes
 * IntrinsicP.h after this file.   Anyway, it doesn't hurt.
 */
#include <X11/CoreP.h>

/*
 * This one is always needed!
 */
#include "BitmapEdit.h"

/* New fields for the BitmapEdit widget class record */

typedef struct {
    int make_compiler_happy;     /* need dummy field */
} BitmapEditClassPart;
```

Example 5-1. BitmapEdiP.h: the class part and class record (continued)

```
/* Full class record declaration */
typedef struct _BitmapEditClassRec {
    CoreClassPart    core_class;
    BitmapEditClassPart    bitmapEdit_class;
} BitmapEditClassRec;
```

Like most widget classes, BitmapEdit provides no new fields in the class part, but it needs one dummy member to make the C compiler happy. Few classes define new class part fields because the Core class already provides all the class fields that Xt knows what to do with.*

The `CorePart` and `CoreClassPart` structures are defined in Core's private header file *<X11/CoreP.h>*. In general, you need to include the private header file of the superclass at the top of your private header file. It turns out that if the header files at the top of the *.c* file are in the right order, and the superclass of your class is Core, Composite, or Constraint (a class defined by Xt), then this header file doesn't need to be included. But it doesn't hurt to do so, because the header files begin with an `ifndef` statement that allows the preprocessor to make sure that no header file is included twice.

If the class structure contains a pointer to an extension structure, you can add to the class structure in later releases of your widget and maintain binary compatibility with subclasses. This feature is used in the basic Intrinsics classes Composite and Constraint, but is unlikely to be useful to you. However, when you do define an extension structure, you do so in the private header file.

5.2.3 Instance Part and Instance Record

The instance record is built exactly like the class structure: by defining new fields in a part structure, and then by combining the instance parts of all superclasses in the instance record. `BitmapEditPart` defines BitmapEdit's new widget instance fields, and the entire widget instance record is `BitmapEditRec`. These structures are defined in *BitmapEdiP.h* (along with the class structures just shown) and are shown in Example 5-2. We've included several likely instance variables, but none of those shown is an essential part of the widget structure.

Example 5-2. BitmapEdiP.h: the instance part and instance record

```
/* New fields for the BitmapEdit widget record */
typedef struct {
    /* resources */
    Pixel  foreground;
    XtCallbackList  callback;/* application installed callback fns */
    Dimension  pixmap_width_in_cells;
    Dimension  pixmap_height_in_cells;
    int  cell_size_in_pixels;
    int  cur_x, cur_y;      /* pstn of visible corner in big pixmap */
```

*Nevertheless, adding new fields does have its uses. If you place a new method field here, subclasses can choose to inherit the function or to replace it. However, the widget code will have to call the method itself because Xt doesn't know when to. This fine point is described in Section 11.4.5.

Example 5-2. BitmapEdiP.h: the instance part and instance record (continued)

```
    char  *cell;              /* for keeping track of array of bits */
    Boolean showAll;          /* whether bitmap should display
                                 entire bitmap */
    /* private state */
    Dimension  pixmap_width_in_pixels;
    Dimension  pixmap_height_in_pixels;
    Pixmap  big_picture;
    GC  draw_gc;              /* for drawing into pixmap */
    GC  undraw_gc;            /* for undrawing into pixmap */
    GC  copy_gc;              /* for copying pixmap into window */
} BitmapEditPart;

/*
 * Full instance record declaration
 */
typedef struct _BitmapEditRec {
    CorePart        core;
    BitmapEditPart    bitmapEdit;
} BitmapEditRec;

#endif                        /* _ORABitmapEditP_h */
```

Unlike the class part, which generally defines no new fields, the instance part of most widgets *does* define new instance variables. These variables control all the configurable elements of the widget, and they hold the widget's state.

Some of these instance variables are resources because they are listed in the resource list in the *BitmapEdit.c* file. (This resource list has exactly the same format as the resource list you saw in the application code in Section 3.5.2). These variables are known as public instance variables (because they are readable and writable from the application). By convention, the public instance variables are placed first in the instance part structure, and comments indicate which fields are public. When the application instantiates the widget, Xt sets the public fields based on the resource databases, the command line (the *-xrm* form), and the argument list passed to `XtVaCreateManagedWidget`. Later on, the application may change these fields using `XtVaSetValues`.

The private instance structure fields (not listed in the widget's resource list) are either derived from resource values or they hold some aspect of the widget's state (such as whether it is highlighted). As in Example 5-2, graphics contexts (GCs) are always found here as private fields if the widget draws any graphics. Graphics contexts are needed for doing drawing, and are derived from public instance structure fields such as colors and fonts, because it is the GC that actually carries color and font information to the X server. GCs were introduced in Section 4.3.1.

You may notice that the `Part` structures of the superclasses are referenced in the `BitmapEditRec` structure. As mentioned earlier, you get these by including the private include file of the immediate superclass, which in turn includes the private include file of its own superclass, and so on.

Both the class and instance structures defined in *BitmapEdiP.h* are `typedef` templates; they do not allocate storage. Xt allocates storage for the instance record when the application calls `XtVaCreateWidget` or `XtVaCreateManagedWidget` to create the widget. The class record, on the other hand, is initialized statically (at compile time) in the *.c* file.

The compiler makes sure that the definition of the class record (in *BitmapEdiP.h*) and the initialization of the class record (in *BitmapEdit.c*) use identical structures, since *BitmapEdiP.h* includes *BitmapEdit.c*. If a field is accidentally left out of the class structure in either file, the compiler will catch the problem (but if the same member is left out of both files, the problem won't be caught and Xt will likely dump core).

BitmapEdiP.h contains an `extern` reference to the class structure initialized in *BitmapEdit.c*. This reference is shown in Example 5-3.

Example 5-3. BitmapEdiP.h: declaring the external class record

```
extern BitmapEditClassRec bitmapEditClassRec;
```

Because the private header file includes the public header file, there is no obvious reason for this `extern` declaration. But since all widget code seems to have it, we go along with the convention.

The naming conventions for the various structure declarations in the private header file are important, and can be confusing. A table summarizing the conventions for types and variables in the widget implementation files is shown in Section 5.6 (after the contents of the *.c* and *.h* files are shown).

That's all there is in the private header file! If you should need to refer back to the private header file for BitmapEdit, it is listed with the rest of the source for the widget in Appendix E, *The xbitmap Application*.

5.3 The Widget Implementation File—BitmapEdit.c

The central element of the *.c* file is the initialization of the class record. Remember that the `typedef` of the class record was declared in *BitmapEdiP.h*, but the record is allocated and the actual values in each field are set in *BitmapEdit.c*. When Xt takes over control of the widgets after the application calls `XtAppMainLoop`, it is the values in this class record that supply Xt with all the information it uses to manage widget instances.

The organization of the *.c* file is quite simple. First, it defines everything that will be placed into the class record, and then initializes the class record, setting fields using these definitions. The major things that need defining are the functions that implement each method, the resource list, the translation table, and the actions table. It would be logical to define these four things at the beginning of the source file, and then put the class record last. This is almost the case, except that by convention the methods and actions are declared at the top of the source file and then defined at the end *after* the class record initialization. This actually makes the widget code clearer because the method declarations provide a complete list of the methods that will be defined later in the file, and the class record remains near the top of the

file where it's easier to find. (Not all the methods have to be defined by a class, because some methods can be inherited or not used, as we will discuss in Section 5.3.6.) Figure 5-2 summarizes the conventional order of code in the *.c* file.*

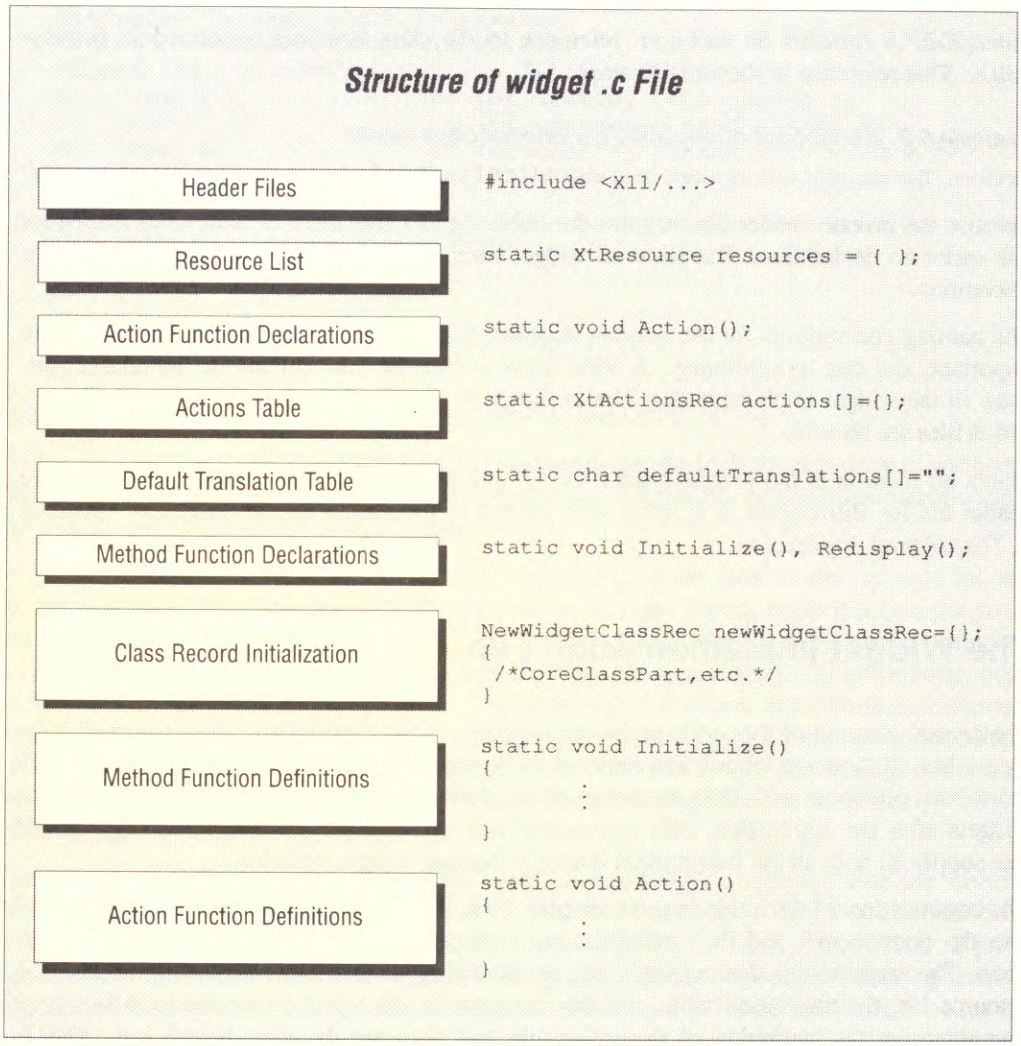


Figure 5-2. Order of code in widget .c file

*Here we use the terminology defined in Kernighan and Ritchie's *The C Programming Language* (Prentice-Hall 1978). A *declaration* announces the properties of a variable (its type, size, etc.), while a *definition* causes storage to be allocated. This distinction is important in the Toolkit because many things are declared and defined in separate steps.

5.3.1 Obligatory Include Files

The *.c* file begins with the standard includes:

- *<stdio.h>*, since `printf` always comes in handy for debugging purposes.
- *<X11/IntrinsicP.h>* (which includes *<X11/Intrinsic.h>*) for the Xt supplied widget classes Core, Composite, and Constraint, declarations of Intrinsics functions, and several useful macros.
- *<X11/StringDefs.h>* for the standard resource names used in defining the resource list.

Example 5-4. BitmapEdit.c: include files

```
#include <stdio.h>

#include <X11/StringDefs.h>
#include <X11/IntrinsicP.h>

#include "BitmapEdiP.h"
```

Remember that *BitmapEdiP.h* includes the private header file of the immediate superclass (which includes the private header file of its own superclass, and so on), and each private header file includes the public header file for its class, so that all the information in all the public and private header files for all superclasses is available to this *.c* file as a result of this one include statement.

We'll look at *BitmapEdit.c* in seven parts, in an order corresponding to Figure 5-3.

5.3.2 Defining the Resource List

A widget inherits the resources defined by its superclasses, and it can also add its own resources by defining a resource list and setting it into the class structure. A widget resource list is identical to an application resource list, which you saw added to *xbitmap* in Chapter 4. The only difference is that a widget need not call `XtGetApplicationResources`.

In creating an application resource list, we created a structure called `app_data` whose fields were to be set through resources. In widget code, the instance part structure is used just like `app_data`, except that the private instance structure fields will not appear in the resource list. Each member of the instance structure that is to be a resource must have an entry in the resource list.

Example 5-5 shows a resource list for the public instance variables defined previously in Example 5-2.

Example 5-5. BitmapEdit's resource list

```
#define offset (field) XtOffsetOf (BitmapEditRec, field)
static XtResource resources[] = {
      {
    XtNforeground,
    XtCForeground,
```

Example 5-5. BitmapEdit's resource list (continued)

```
    XtRPixel,
    sizeof(Pixel),
    offset(bitmapEdit.foreground),
    XtRString,
    XtDefaultForeground
     },
     {
    XtNcallback,
    XtCCallback,
    XtRCallback,
    sizeof(XtPointer),
    offset(bitmapEdit.callback),
    XtRCallback,
    NULL
     },
     {
    XtNcellSizeInPixels,
    XtCCellSizeInPixels,
    XtRInt, sizeof(int),
    offset(bitmapEdit.cell_size_in_pixels),
    XtRImmediate,
    (XtPointer)DEFAULT_CELL_SIZE
     },
     {
    XtNpixmapWidthInCells,
    XtCPixmapWidthInCells,
    XtRDimension,
    sizeof(Dimension),
    offset(bitmapEdit.pixmap_width_in_cells),
    XtRImmediate,
    (XtPointer)DEFAULT_PIXMAP_WIDTH
     },
     {
    XtNpixmapHeightInCells,
    XtCPixmapHeightInCells,
    XtRDimension,
    sizeof(Dimension),
    offset(bitmapEdit.pixmap_height_in_cells),
    XtRImmediate,
    (XtPointer)DEFAULT_PIXMAP_HEIGHT
     },
     {
    XtNcurX,
    XtCCurX,
    XtRInt,
    sizeof(int),
    offset(bitmapEdit.cur_x),
    XtRImmediate,
    (XtPointer) 0
     },
     {
    XtNcurY,
    XtCCurY,
    XtRInt,
    sizeof(int),
    offset(bitmapEdit.cur_y),
```

Example 5-5. BitmapEdit's resource list (continued)

```
    XtRString,
    (XtPointer) NULL
     },
     {
    XtNcellArray,
    XtCCellArray,
    XtRString,
    sizeof(String),
    offset(bitmapEdit.cell),
    XtRImmediate,
    (XtPointer) 0
     },
     {
    XtNshowEntireBitmap,
    XtCShowEntireBitmap,
    XtRBoolean,
    sizeof(Boolean),
    offset(bitmapEdit.showAll),
    XtRImmediate,
    (XtPointer) TRUE
     },
};
```

The details of each field in a resource list entry will be presented in Chapter 9, *Resource Management and Type Conversion*.

As mentioned earlier, a widget class inherits all the resources defined in the resource lists of its superclasses. If a resource is given the same name as a superclass resource, it overrides the superclass resource. The only reason to create a new resource with the same name as a superclass resource is to give it a new, subclass-specific default value. The type and offset should remain the same.

When defining a resource list, use as many as possible of the constants defined in *<X11/StringDefs.h>*. However, any constant unique to this widget class can be defined in the public include file, as will be described in the next section.

Table 5-1 summarizes the conventions for the constants used in the resource list.

Table 5-1. Resource List Constant Conventions

Prefix	First word capitalization	Description
`XtN`	Lower	Resource name
`XtC`	Upper	Resource class
`XtR`	Upper	Representation type

The representation type of a resource is a string that represents the type in which the widget stores the resource value. Xt constants and functions of all types use upper case letters whenever a word break might otherwise be called for. Two examples of resource names following this convention are `XtNborderColor` and `XtNmappedWhenManaged`.

Example 5-6 shows how the resource list is entered into the class structure. You'll see this again in Section 5.3.5 where we show the entire class record and describe how to initialize it. The names of the fields in the class structure are shown in the comments at right.

Example 5-6. Setting the resource list into the class structure

```
BitmapEditClassRec bitmapEditClassRec = {
    {  /* Core class part */
          .
          .
        resources,              /* resources */
        XtNumber(resources),    /* resource_count */
          .
          .
    },
    {  /* BitmapEdit class part */
        0,                      /* dummy_field */
    }
};
```

Note that a widget or an application can get the resource list for a class using `XtGetResourceList`, which returns a pointer to a resource list of the same form as shown above. This function isn't needed in most applications or widgets.

5.3.3 The Translation Table and Actions Table

You may recall that the translation table maps event sequences into string action names, and the action table then maps string action names into actual action functions. This is done in two steps so that the translation table is a single string that can be specified in the resource databases (since resource databases are always composed entirely of strings). The action table is not configurable through the resource database, but it can be added to the application or with `XtAppAddAction(s)`.

Like the resource list, the default translation table and the actions table have to be defined before the class record can be initialized. They determine which events this widget will respond to, and which functions defined in this source file will be triggered by those events.

The translation table is a resource defined by the Core class. It is a strange resource in several ways. It has no default value in the resource list. Instead, the default translation table is initialized into the class structure. Translations are not inherited like other resources—you do not get the sum of all the translation tables registered by the superclasses. Rather, the translation table you specify in the class structure is the only one (in the code) that matters. However, you can *choose* to use the immediate superclass's translation table instead of defining one, in which case you initialize the translation table field in the class structure to be `XtInheritTranslations`.

Each widget has its own action list, and if the application registers an action list, it is kept as a separate list. When an event combination occurs in a widget, Xt translates the event combination into an action string and searches that widget's action list. If the action string is found in the widget's action list, the search stops and that action function is called. If the action string is not found in the widget's action list, then the application's action list is searched. If

neither list contains the appropriate action string, Xt prints a diagnostic warning. If the application and the widget both define the same action string in the actions table, the application's function mapped to that string will never be called. Two widget classes, however, may have the same action function name, and they will not conflict.

Example 5-7 shows a default translation table and an action table. Note that, like methods, the action functions are declared before being used in the actions table, but they will actually be defined later in the source file.

Example 5-7. The default translation table and the actions table

```
static void DrawCell(), UndrawCell(), ToggleCell();

static char defaultTranslations[] =
    "<Btn1Down>:    DrawCell()              \n\
     <Btn2Down>:    UndrawCell()            \n\
     <Btn3Down>:    ToggleCell()            \n\
     <Btn1Motion>:  DrawCell()              \n\
     <Btn2Motion>:  UndrawCell()            \n\
     <Btn3Motion>:  ToggleCell()";

static XtActionsRec actions[] = {
    {"DrawCell", DrawCell},
    {"UndrawCell", UndrawCell},
    {"ToggleCell", ToggleCell},
};
```

The pointers to the actions table and translation table are placed into the class structure just like the resource list (but of course into different fields), as shown in Example 5-8. The names of the fields in the class structure are shown in the comments at right. (You'll see this again shortly in the section on class structure initialization.)

Example 5-8. Translations in the Core class record

```
BitmapEditClassRec bitmapEditClassRec = {
    {  /* core class part */
        .
        .
        actions,              /* actions */
        XtNumber(actions),    /* num_actions */
        .
        .
    defaultTranslations,      /* tm_table */
    },
    {                         /* BitmapEdit class part */
    0,                        /* dummy_field */
    }
};
```

Note that the default translation table cannot be compiled with `XtParseTranslationTable` before being placed in the class structure, since the class structure initialization occurs at compile time. Xt compiles the default translations when the class is initialized.

The translation table and actions table are discussed more fully in Chapter 7, *Events, Translations, and Accelerators*.

5.3.4 Declaring Methods

Xt calls a method when the application calls a certain Xt function. For example, when an application creates a widget with `XtVaCreateManagedWidget` (or any similar call), Xt calls the `initialize` method of that widget class. There are separate methods called when `Expose` events occur, when `XtVaSetValues` is called, and when `XtDestroyWidget` is called. Methods are distinct from actions in that methods are called in response to application function calls and `Expose` events, while actions are usually called in response to user-selectable events. Methods also each have their own field in the class structure, while actions are listed in a table that is stored in one field of the class structure. Pointers to action functions are placed in an action table which is then entered into the class structure, while a pointer to each method function is entered directly into its field of the class structure.

The widget's methods should be declared near the top of the *.c* file. This is so that the class structure initialization can appear before the method definitions. As usual in C, they should be declared as the actual type returned, such as `void` or `Bool`.* They should be declared static so that the scope of these variables is limited to this source file, eliminating possible conflicts with other widget classes. Example 5-9 shows the method declarations from `BitmapEdit`.

Example 5-9. BitmapEdit.c: function type declarations

```
/* Declaration of methods */

static void Initialize();
static void Redisplay();
static void Destroy();
static void Resize();
static Boolean SetValues();
static XtGeometryResult QueryGeometry();

/* these Core methods not needed by BitmapEdit:
 *
 * static void ClassInitialize();
 * static void Realize();
 */

/* the following are functions private to BitmapEdit */
static void DrawPixmaps(), DoCell(), ChangeCellSize();

/* the following are actions of BitmapEdit */
static void DrawCell(), UndrawCell(), ToggleCell();
```

*Note that you cannot declare methods using the prototype procedure symbols Xt defines such as `XtExposeProc` (for the `expose` method), since these are defined to be pointers to functions that return the right type; they are not return types. Their only use is within Xt.

5.3.5 Initializing the Class Record

We've already shown you how to create the resource list, the translation table, and the actions table, and set into the class structure. A major part of the work is done. Now we just need to insert the method names we declared earlier into the class record, and set a few of the miscellaneous data fields.

As we saw in the discussion of the private header file, the BitmapEdit class record includes class parts for BitmapEdit itself and for each superclass of BitmapEdit, in this case only Core. To initialize the class record, each class part has to be initialized field by field. But for simple (noncomposite) widgets this job comes down to initializing the Core class part, because few classes actually define new class part fields.

5.3.5.1 The Core Class Part

Since all widget classes are subclasses of Core, all need to initialize the Core class part. Example 5-10 shows the core part initialized as needed by the BitmapEdit widget. The actual name of each field is shown in the comment at left. Each field will be discussed after the example.

Example 5-10. BitmapEdit.c: initialization of Core class record

```
BitmapEditClassRec bitmapEditClassRec = {
    {
    /* core_class fields */
    /* superclass     */            (WidgetClass) &coreClassRec,
    /* class_name     */            "BitmapEdit",
    /* widget_size    */            sizeof(BitmapEditRec),
    /* class_initialize    */       NULL,
    /* class_part_initialize    */ NULL,
    /* class_inited    */           FALSE,
    /* initialize     */            Initialize,
    /* initialize_hook    */       NULL,
    /* realize        */            XtInheritRealize,
    /* actions        */            actions,
    /* num_actions    */            XtNumber(actions),
    /* resources      */            resources,
    /* num_resources     */         XtNumber(resources),
    /* xrm_class      */            NULLQUARK,
    /* compress_motion     */       TRUE,
    /* compress_exposure     */     TRUE,
    /* compress_enterleave     */   TRUE,
    /* visible_interest     */      FALSE,
    /* destroy     */               Destroy,
    /* resize      */               Resize,
    /* expose      */               Redisplay,
    /* set_values     */           SetValues,
    /* set_values_hook     */      NULL,
    /* set_values_almost     */    XtInheritSetValuesAlmost,
    /* get_values_hook     */      NULL,
    /* accept_focus     */         NULL,
    /* version     */               XtVersion,
    /* callback_private     */     NULL,
```

Example 5-10. BitmapEdit.c: initialization of Core class record (continued)

```
    /* tm_table     */              defaultTranslations,
    /* query_geometry     */       QueryGeometry,
    /* display_accelerator     */  XtInheritDisplayAccelerator,
    /* extension     */            NULL
    },
    { /* BitmapEdit class part*/
    /* dummy_field     */ 0,
    },
};
```

If you are like most programmers, the core class structure is the biggest structure you have ever seen! Don't worry, because many of the fields you will never have to worry about, and the rest you will gradually come to know as you need them. We will introduce all the fields here, but you are not expected to absorb all the details in a single sitting. Treat this section both as an introduction and as a summary to which you can turn back when you encounter a field you don't understand. Also, all of the methods and some of the data fields will be described in more detail later in the book. These field descriptions reference the section in this book where you will find additional information about the field.

- The `superclass` field is set to a pointer to the superclass's class structure. This defines which widgets this class can inherit from. For a subclass of Core this would be `&coreClassRec` (`widgetClassRec` was used in previous releases); for a subclass of Composite, `&compositeClassRec`; for a subclass of Constraint, `&constraintClassRec`, and so on.
- The next field, `class_name`, contains the name that will be used to set resources by class. In other words, this is the string that you want to appear in the resource database when setting resources for all instances of this class.
- The `widget_size` field is the size of the instance record. This should always be specified using `sizeof` with the complete instance record declaration for this class, defined in *BitmapEdiP.h*, as an argument. In this case the field is initialized to `sizeof(BitmapEditRec)`. Xt uses this field to allocate memory for instance records at run-time.
- The next field, `class_initialize`, is the first of many pointers to widget methods. There are several issues regarding methods that require separate treatment, so we'll describe these in the next section. The complete list of methods in the Core part structure is as follows: `class_initialize`, `class_part_init`, `initialize`, `realize`, `destroy`, `resize`, `expose`, `set_values`, `set_values_almost`, `query_geometry`, and `accept_focus`.
- The `display_accelerator` field is used in conjunction with accelerators, which are a way of redirecting events to actions in different widgets, and will be discussed in more detail in Chapter 9, *Resource Management and Type Conversion*.
- The `class_inited` field is used internally by Xt to indicate whether this class has been initialized before. Always initialize it to `FALSE`.
- The `initialize_hook`, `set_values_hook`, and `get_values_hook` fields are for use in widgets that have `get_values_hook` fields is for use in widgets that

have subparts that are not widgets. Subparts can have their own resources, and can load them from the resource databases as a widget can. This method is called immediately after the `get_values` method, and is for performing the same operations except on subparts. This field is described in Chapter 9, *Resource Management and Type Conversion*. (Most widgets that used subparts have now been converted to use non-widget objects or gadgets instead.)

- The fields relating to resources, the default translation table, and the actions table have already been described. The only one of these fields without an obvious name is the `tm_table` field, in which you place the default translation table.
- The `xrm_class` field is used internally by Xt, and must always be initialized to `NULLQUARK`. This is a fixed initialization value.
- The `compress_motion`, `compress_exposure`, and `compress_enterleave` fields control the Toolkit's *event filters*. Basically, these filters remove events that some widgets can do without, thus improving performance. Unless a widget performs complicated drawing or tracks the pointer, `compress_exposure` should usually be `XtExposeCompressMultiple` and the other two fields should usually be `TRUE`. These filters are described in Chapter 8, *Other Input Sources*.
- The `visible_interest` field can be set to `TRUE` if your widget wishes to get `VisibilityNotify` events, which signal changes in the visibility of your widget. Normally this is set to `FALSE`, because `Expose` events cause the widget to be redrawn at the proper times. For some widgets, however, `Expose` events are not enough. If your widget draws continuously, as in a game, it can stop computing output for areas that are no longer visible. There are other cases where `VisibilityNotify` events are useful.
- Xt uses the `version` field to check the compiled widget code against the library the widget is linked against. If you specify the constant `XtVersion` and it is different from the version used by the libraries, then Xt displays a run-time warning message. However, if you have intentionally designed a widget to run under more than one version of Xt, you can specify the constant `XtVersionDontCheck`.
- The `callback_private` field is private to Xt, and you always initialize it to `NULL`.
- The `extension` field is for later expansion of the widget while maintaining binary compatibility. When used, it is a pointer to an extension structure containing additional class structure fields.

5.3.5.2 Initializing the Core Methods

As you've just seen, there are several places for pointers to methods stored in the Core class structure. We will be describing the purpose of each of these methods in the next section. But first, a word about how to set the method fields.

Fortunately, the Core class already defines some of the methods, and you can choose to inherit them instead of writing your own. In general, when you set out to write a widget, you will pick as your superclass the widget that has the most methods that you can inherit instead of writing from scratch.

Broadly speaking, there are two types of methods, self-contained methods and chained methods.

A *self-contained* method is one that is called alone—the methods of the same name in the widget's superclasses are not called. For example, `expose` is self-contained. When you write the `expose` method, you are writing all the drawing code for the widget. The `expose` method of the superclass will not be called even if it was designed to do drawing. Therefore, by writing an `expose` method you replace the `expose` method of the superclass.

A *chained method* is one that is not called alone—it is called either before or after all the methods of the same name in its superclasses and subclasses. Therefore, you can't replace any of the code in this method in the subclasses or superclasses, you can only add to it by writing your own.

Inheritance works differently for chained and self-contained methods. For self-contained methods, such as `realize` and `expose`, inheritance works almost as it does for default translations. When initializing a field in the class record that represents a self-contained method, you have three choices:

- You can define these methods in your widget code by placing the name of the function in the class record and defining that function somewhere in the source file.
- You can inherit that method from the immediate superclass by placing a special symbol beginning with `XtInherit` in that field in the class record.
- You can use the first technique, but reference the superclass's method in your method. This allows you to add features to the superclass's method without having to completely copy it and modify it.

These techniques are demonstrated in Section 5.3.8. Chained methods, on the other hand, use one of two flavors of inheritance. Some are *downward chained*, which means that the Core method is called first, followed by the same method in each subclass. Other methods are *upward chained*, which means that the Core class method is called last. For both upward and downward chained methods, your choice is whether to specify additional code by defining your own method, or to go with what the other classes have already defined. You cannot prevent the code from the other classes from being executed.

Here is how chaining works for the `initialize` method in BitmapEdit. Remember that BitmapEdit is a subclass of Core. The Core widget's class record contains only the Core part structure. The BitmapEdit widget's class record contains the Core and BitmapEdit part structures. The `initialize` method is present in the Core part structure of both classes. When an application creates an instance of the BitmapEdit widget, Xt calls the function specified in the `initialize` field in the Core class record first, followed by the one in the BitmapEdit class record. This is an example of downward chaining.

The `destroy` method, on the other hand, is upward chained. That is, the `destroy` method in BitmapEdit would be called first, followed by the `destroy` method for Core.

If you specify `NULL` in your class structure for a method that chains upward or downward, it is equivalent to specifying a function that does nothing. The function for that method of the superclasses and subclasses will still be called normally.

Table 5-2 lists which methods fall into each type of inheritance. It also shows how translations, actions, and resources are chained.

Table 5-2. Inheritance Style of Various Methods

Self-contained	Upward chained	Downward chained
`class_initialize`	`destroy`	`class_part_init`
`realize`		`initialize`
`resize`		`set_values`
`expose`		`get_values_hook`
`accept_focus`		`set_values_hook`
`set_values_almost`		`initialize_hook`
`query_geometry`		
translations	actions	resources

5.3.6 Description of Core Methods

Here is a brief description of the purpose of each Core method, and where in this book the method will be described in detail. All these methods, except `realize`, can be set to `NULL` in the class record and the widget will still function. However, all widgets that draw into their window will also require the `expose` method, the `initialize` method, and usually the `resize` method. And to be good children, widgets should define a `query_geometry` method. The `realize` method is shown in Section 5.3.8, and the rest of the commonly-used methods immediately after that in Chapter 6, *Basic Widget Methods*.

This list describes *all* the methods, even those that are rarely used. There is a lot of detail here that you should not expect to absorb in a first reading. Like the list of Core class structure fields, treat this as an introduction and come back to it later for reference when you come across a method you don't know how to use, or if you have something you want to do and you don't remember in which method to do it.

- `initialize` sets initial values for all the fields in the instance part structure. This method is responsible for checking that all public fields have been set to reasonable values. This method is downward chained, so each class's `initialize` method sets the initial values for its own instance part structure. This method is described in Chapter 6, *Basic Widget Methods*.
- `initialize_hook` is called immediately after the initialize method of the same class. It is now obsolete (though still called for compatibility); its job has been added to that of `initialize`. It allows the widget to initialize subparts, and is used only in widgets that have subparts. Subparts have their own resources and are described in Section 9.4. `initialize_hook`, a downward chained method, is also described there.

- `class_initialize` is called once, the first time an instance of a class is created by the application. The widget registers type converters here, if it has defined any nonstandard ones. `class_initialize` is self-sufficient and is described in Chapter 9, *Resource Management and Type Conversion*.

- `class_part_init` is called once the first time an instance of a class is created by the application. It is different from `class_initialize` only in that it is downward chained. This method resolves inheritance of self-sufficient methods from the immediate superclass. It is needed only in classes that define their own methods in their class part (but is not present in Core, Composite, or Constraint, because Xt handles inheritance in these). This method is described in Chapter 13, *Miscellaneous Toolkit Programming Techniques*.

- `realize` is called when the application calls `XtRealizeWidget`. This method is responsible for setting window attributes and for creating the window for the widget. It is self-sufficient, and is described in Section 5.3.8.

- `expose` redraws a widget whenever an `Expose` event arrives from the server (but note that Xt can coalesce consecutive `Expose` events to minimize the number of times it is called). This method is responsible for making Xlib calls to draw in the widget's window. The widget's instance variables are often used in the `expose` method to guide the drawing. This method is self-sufficient. This method is described in Chapter 6, *Basic Widget Methods*.

- `resize` is called when the parent widget resizes the widget. It recalculates the instance variables based on the new position and size of its window, which are passed into the method. This method is self-sufficient and is described in Chapter 6, *Basic Widget Methods*, and in Chapter 11, *Geometry Management*.

- `set_values` is called whenever the application calls `XtSetValues` to set the resources of the widget. This method recalculates private instance variables based on the new public instance variable values. It contains similar code to the `initialize` method, but is called at different, and perhaps multiple, times. The `set_values` method is downward chained. This method is described in Chapter 6, *Basic Widget Methods*.

- `set_values_almost` is used to process application requests to change this widget's size. This field should never be `NULL`. Unless you've written your own `set_values_almost` method, this field should be set to `XtInheritSetValuesAlmost`. Most classes inherit this procedure from their superclass. This method is self-contained and is described in Chapter 11, *Geometry Management*.

- `set_values_hook` sets resource values in subparts. It is now obsolete (though still called for compatibility); its job has been added to that of `set_values`. This method is used only in widgets that have subparts, as described in Section 9.4. It is downward chained and is described in Chapter 9, *Resource Management and Type Conversion*.

- `accept_focus` is `NULL` for most widgets (or, at least, for all the Athena widgets). When it is present, this method should set the keyboard focus to a subwidget of this widget. This would be used, for example, to allow the application to set the input focus to the Text widget within a Dialog widget. This method is invoked when the application calls

calls `XtCallAcceptFocus`. This method is self-contained and is described in Chapter 13, *Miscellaneous Toolkit Programming Techniques*.

- `get_values_hook` is called just after `get_values` and is used to return the resources of subparts. This method is downward chained and is described in Chapter 9, *Resource Management and Type Conversion*.
- `destroy` deallocates local and server memory allocated by this widget. This is called when an application destroys a widget but remains running. This method is described in Chapter 6, *Basic Widget Methods*.
- `query_geometry` may be called when the parent widget is about to resize the widget. The method is passed the proposed new size, and is allowed to suggest a compromise size, or to agree to the change as specified. This method it self-contained. It is described in Chapter 6, *Basic Widget Methods*.

Initialization of the Composite and Constraint class parts, including the methods in those structures, is described in Chapter 11, *Geometry Management*, since this is necessary only in widgets that manage children.

5.3.7 Packaging the Class Record for Application Use

The final requirement of the *.c* file is a pointer to the class record, called `bitmapEditWidgetClass`, that applications use as an argument to `XtCreateManagedWidget` to create instances of this widget class. This is shown in Example 5-11.

Example 5-11. BitmapEdit.c: declaring the class record pointer

```
WidgetClass bitmapEditWidgetClass = (WidgetClass) &bitmapEditClassRec;
```

`bitmapEditWidgetClass` is set to be a pointer to the `bitmapEditClassRec`, the complete class record. Remember that since the actual declaration of the class structure is in the private include file, the application cannot access class structure fields, and therefore this pointer is opaque to the application.

5.3.8 A Sample Method

Each method has a particular job to do, and will be described in the chapter that discusses that job. However, we'll describe the `realize` method now, because it is simple and demonstrates the two techniques of inheriting for self-contained fields that were described in Section 5.3.5.2. The `realize` method is responsible for creating the widget's window, and all widgets have a window.* The first technique is to inherit wholesale the method defined by the immediate superclass. The superclass may have its own `realize` method, or it may also inherit the method from its superclass, and so on. The Core widget's `realize` method

*A windowless widget is called a gadget or an object. Gadgets and objects have less features than widgets, but are also less costly in server memory and network traffic. Gadgets are described in Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*, and objects in Chapter 13, *Miscellaneous Toolkit Programming Techniques*.

creates a basic window.* In your subclasses of any of these widgets, you can inherit the Core `realize` method without modification by initializing the `realize` member of the Core class record to the symbolic constant `XtInheritRealize`, as shown in Example 5-12.

Example 5-12. BitmapEdit.c: inheriting a self-contained method

```
BitmapEditClassRec bitmapEditClassRec = {
    {                   /* Core class part */
        .
        .
        .
    XtInheritRealize,   /* realize */
        .
        .
        .
    },
    {   /* BitmapEdit class part */
    0,                  / * dummy_field */
    }
};
```

Xt also defines symbolic constants for inheriting for every other self-contained method. They all begin with `XtInherit`, and in general continue with the capitalized name of the method field.

An important part of the process of creating an X window is the setting of window attributes. Therefore, a brief aside on window attributes is necessary. (You can skim the next page if you are already familiar with them.)

Window attributes are basic features of the way the server makes windows look and act. Window attributes can also be changed later if necessary, but the `realize` method is the place to set them initially. The Core class `realize` method sets some basic window attributes such as the window background and border colors or patterns, and it gets these values from Core resources. Here is a list of the window attributes that you may wish to set:

Background	Can be a solid color, pattern, or transparent.
Border	Can be a solid color or pattern.
Bit Gravity	Determines how partial window contents are preserved when a window is resized. This is an optimization that can save redrawing.
Backing Store	Provides hints about when a window's contents should be preserved by the server even when the window is obscured or unmapped. This is useful for widgets that are very time-consuming to redraw. Not all servers are capable of maintaining a backing store. Check the value returned from the

*If you have the source code for Xt, you may discover that the Core widget is actually an amalgamation of several other superclasses called WindowObj, RectObj, and so on. (You might also notice the include files for these classes in */usr/include/X11*.) The Core widget actually inherits the `realize` procedure from WindowObj. These classes are an implementation detail that will affect you only if you want to look in some of these superclasses to find the code that implements features normally attributed to Core. From application code or widget code, it is always safe to assume that Core is the top of the class tree. For gadgets, you need to know more about these "hidden" classes, so we'll discuss them in Chapter 12.

Xlib `DoesBackingStore` macro to determine whether this feature is supported on a particular screen on your server.

Saving Under — Provides hints about whether or not the screen area beneath a window should be saved while a window such as a pop-up menu is in place, to save obscured windows from having to redraw themselves when the pop up is removed. Not all servers can save under windows. You can find out whether this feature is supported on a particular screen with the Xlib `DoesSaveUnders` macro.

Colormap — Determines which virtual colormap should be used for this window. If your widget requires a lot of specific colors—for example, to draw a shaded image, it may need to create its own virtual colormap. In that case, it would set this attribute to the ID of the created colormap. For more information, see Chapter 7, *Color*, in Volume One, *Xlib Programming Manual*.

Cursor — Determines which cursor should be displayed when the pointer is in this window. You must create this cursor before setting this attribute. This can be done with a standard type converter, as described in Chapter 13, *Miscellaneous Toolkit Programming Techniques*.

It may clarify the picture to describe the features that window attributes *do not* affect. Setting the window attributes does not determine a window's parent, depth, or visual. These are all set when a window is created, and are permanent. The window attributes are also not used for setting the size, position, or border width of a widget. These are set using `XtSetValues`. Window attributes do not determine how graphics requests are interpreted; this is the job of the graphics context (GC).

Note that some of the window attributes are not listed here because they should not be set directly by widgets, in the `realize` method or anywhere else. These include the `event_mask`, which controls which events are sent to this widget. Xt itself sets this window attribute based on the translation table. Another is `override_redirect`, which is handled by the Shell widget.

You could write a `realize` method that set your desired attributes and then called `XtCreateWindow`. But this is slightly wasteful, since Core already has a `realize` method that creates a window and you can take advantage of it. This is the second inheritance scheme used for self-contained fields. You define your own `realize` method just as if you were going to write it from scratch, but then you call the superclass's realize method directly, as shown in Example 5-13.

Example 5-13. Inheriting by invoking the superclass method from a widget method

```
#define superclass      (&coreClassRec)

static void Realize(w, valueMask, attributes)
Widget w;
XtValueMask *valueMask;
XSetWindowAttributes *attributes;
{
    /* this is already set, but just for example */
    *valueMask |= CWBitGravity;
```

Example 5-13. Inheriting by invoking the superclass method from a widget method (continued)

```
    attributes->bit_gravity = NorthWestGravity;

    /* use realize method from superclass */
    (*superclass->core_class.realize) (w, valueMask, attributes);
}
```

Xt passes to the `realize` method a set of window attributes based on Core instance structure values. You update these values as necessary, and then call the superclass's `realize` method, as shown.

See the `XtRealizeProc` reference page in Volume Five, *X Toolkit Intrinsics Reference Manual* to find out the default settings of the window attributes as passed into the `realize` method.

You may wonder what happens if the superclass also inherited its `realize` method—does the code in Example 5-13 crash by assuming the superclass field contains a function pointer when it actually contains the constant `XtInheritRealize`? No. When the class is initialized, Xt reconciles all the inherited methods and resets the class record fields to be pointers to the right methods.

What is happening behind the scenes when you inherit the `realize` method is that the Core `realize` method calls the Toolkit function `XtCreateWindow`, which in turn calls the Xlib function `XCreateWindow`. You may need to call the Xlib routine yourself in the `realize` method if you want to use a visual other than the default. See the reference page for `XCreateWindow` in Volume Two, *Xlib Reference Manual*, and Chapter 7, *Color*, in Volume One, *Xlib Programming Manual*, for details on depth, visual, and the issue of color in general.

5.4 The Public Header File—BitmapEdit.h

The public header file defines the aspects of the widget that can be accessed from the application. Public header files tend to be short. The two obligatory features of the *BitmapEdit.h* file are:

- An external declaration of `bitmapEditWidgetClass`, the class record pointer used by applications in calls to `XtCreateWidget` to create an instance of this widget class.
- A pointer to the widget instance record, in this case `BitmapEditWidget`. Xt calls all methods and actions with an argument of type `Widget`. To access any of the fields in the instance structure, this pointer must be cast to type `BitmapEditWidget`. The easiest way to do this is to declare the argument of the method as type `BitmapEditWidget` (the other way is to cast in an assignment).

If your resource list uses `XtN`, `XtC`, or `XtR` constants not defined in *<X11/StringDefs.h>*, you must define them in the public include file. The definitions should have only a single

space between the definition and the value, with no trailing comment or space. This reduces the possibility of compiler warnings from similar but not identical definitions in multiple classes.

If a widget offers any public functions, they would be declared `extern` here (and actually defined in the *.c* file). Public functions allow the application to read or change certain private data in certain more restricted but more convenient ways than is possible with resources. For example, the BitmapEdit widget provides the public function `BitmapEditGetArrayString` that applications can call to get the array of bits currently stored as private data in the widget. (This array is not an attribute readable with `XtGetValues` because the current implementation does not allow the application to set this array. A future iteration could add this feature, and then the public function would cease to be necessary, though it might be kept anyway for convenience and backward compatibility.)

Example 5-14 shows BitmapEdit's public header file.

Example 5-14. BitmapEdit.h: incidental declarations

```
#ifndef _ORABitmapEdit_h
#define _ORABitmapEdit_h

/*
 * BitmapEdit Widget public include file
 */

/*
 * This include not needed unless the application includes
 * Intrinsic.h after this file.   Anyway, it doesn't hurt.
 */
#include <X11/Core.h>

/* Resources:
 * Name              Class         RepType     Default Value
 * ----              -----         -------     -------------
 * (from RectObj)
 * ancestorSensitive
 * x                 Position      Int         0
 * y                 Position      Int         0
 * width             Dimension     Dimension   0
 * height            Dimension     Dimension   0
 * borderWidth       BorderWidth   Int
 * sensitive         Sensitive
 *
 * (from WindowObj)
 * screen            Screen        Pointer     XtCopyScreen
 * depth             Depth         Int         XtCopyFromParent
 * colormap          Colormap      Pointer     XtCopyFromParent
 * background        Background    Pixel       White
 * backgroundPixmap  Pixmap        Pixmap      XtUnspecifiedPixmap
 * borderColor       BorderColor   Pixel       Black
 * borderPixmap      BorderPixmap  Pixmap      XtUnspecifiedPixmap
 * mappedWhenManaged MappedWhenManaged Boolean True
 * translations
 * accelerators
 *
 * (from Core)
 * none
```

Example 5-14. BitmapEdit.h: incidental declarations (continued)

```
 *
 * (from BitmapEdit)
 * foregroundPixel  Foreground     Pixel      Black
 * backgroundPixel  Background     Pixel      White
 * callback         Callback       Callback   NULL
 * cellSize         CellSize       Int        30
 * pixmapWidth      PixmapWidth    Int        32
 * pixmapHeight     PixmapHeight   Int        32
 */

/*
 * This public structure is used as call_data to the callback.
 * It passes the x, y position of the cell toggled (in units of
 * cells, not pixels) and a mode flag that indicates whether the
 * cell was turned on (1) or off (0).
 */
typedef struct {
    int mode;
    int newx;
    int newy;
} BitmapEditPointInfo;

#define XtNcellSizeInPixels "cellSizeInPixels"
#define XtNpixmapWidthInCells "pixmapWidthInCells"
#define XtNpixmapHeightInCells "pixmapHeightInCells"
#define XtNcurX "curX"
#define XtNcurY "curY"
#define XtNcellArray "cellArray"
#define XtNshowEntireBitmap "showEntireBitmap"

#define XtCCellSizeInPixels "CellSizeInPixels"
#define XtCPixmapWidthInCells "PixmapWidthInCells"
#define XtCPixmapHeightInCells "PixmapHeightInCells"
#define XtCCurX "CurX"
#define XtCCurY "CurY"
#define XtCCellArray "CellArray"
#define XtCShowEntireBitmap "ShowEntireBitmap"

extern char *BitmapEditGetArrayString(); /* w */
       /* Widget w; */
/* Class record constants */

extern WidgetClass bitmapEditWidgetClass;

typedef struct _BitmapEditClassRec *BitmapEditWidgetClass;
typedef struct _BitmapEditRec      *BitmapEditWidget;

#endif /* _ORABitmapEdit_h */
/* DON'T ADD STUFF AFTER THIS #endif */
```

It is a good idea to comment the types of the resources for all superclasses as shown in Example 5-14. Not only is this a good summary for you of all the resources of your widget, but it allows the application programmer to use this file for documentation when creating argument lists.

5.5 The Process of Widget Writing

The process of writing a widget always begins with the same steps. They are:

- Copy all three files of the widget most similar to the one you intend to write; pick one that has many methods defined so that you don't need to type them in. (It's easier to delete than to retype.)
- Globally change the widget class name in the files. The fastest way to do this under UNIX is with *sed*, using a script similar to the following:

  ```
  s/BitmapEdit/NewName/g
  s/bitmapEdit/newName/g
  ```

  Place this script in the file *sedscr*, and run the command:

  ```
  spike% sed -f sedscr file > newfile
  ```

  on each file. (Or write a simple `for` loop to run it on multiple files.)
- Start from the top of the *.c* file, and begin by writing the resource list. While writing the resource list, you may need to edit the public header file to define new resource names and classes (`XtN` and `XtC` symbols). While writing the resource list (and during the entire widget-writing process), you will also need to edit the private header file in order to add and remove instance part structure fields, as you determine a need for them while writing methods and actions. Later you will probably discover additional parameters that you want to define as resources.
- Design the output you expect your widget to draw. Your instance part structure fields must hold all the information necessary to redraw everything when the window is exposed—add the necessary fields.
- Design the user input you expect your widget to accept. Start with as many separate actions as you can—one for each distinguishable user input idiom. For example, BitmapEdit has three actions for changing one bitmap cell: `DrawCell`, `UndrawCell`, and `ToggleCell`. Even though these invoke almost identical underlying code, it is best to keep them as separate actions.
- Design a default translation table to have these actions called in response to the appropriate events. Help in this area is available in Section 7.1.2.
- Write the `expose` method and actions. Just how to do this is described in the next chapter. But in summary, both the `expose` method and actions often draw into the widget. The `expose` method must always be able to redraw what the actions drew. Therefore, it usually pays to have common elements of code called by both an action and the `expose` method. Neither the action nor the `expose` method pass arguments to this common code. Instead, the actions set instance variables that are read in the `expose` method or common code. The instance variables act as global variables because they are available almost everywhere in the widget code.
- Add parameters that allow your drawing code to work smoothly in any size window, and add a `resize` method that sets these parameters.

- Add the `initialize` method to check resource values that might have been user supplied and to initialize private instance variables.
- Add the `set_values` method to check application-supplied resource values and reset private instance variables based on the new resource values.
- Declare the methods and actions you have defined, near the top of the *.c* file.
- Enter all these functions and tables you have defined into the class structure initialization.

It is useful to have a simple application available for testing your widget as you develop it. One that simply creates the widget under construction and provides a Quit button is quite adequate. Then you can add code to the widget incrementally, assuring at each step that the program compiles, links, and runs without error.

As this list implies, it is a good idea to start simply by completing all the above steps for a small subset of the features you eventually want. Once you have a working widget that you can test, you can add features one at a time by going through the list again. If instead you attempt to write an ambitious widget in one pass, you will spend much longer debugging it.

Once you have learned how to write methods and actions in the next chapter, *Basic Widget Methods*, you should be ready to write a simple widget.

5.6 Summary of Conventions

The naming conventions for the various structure declarations in the widget source files can be confusing. Table 5-3 summarizes these conventions, using the BitmapEdit widget as an example, and describes where each type is used. This table is just to help you read the code. If you create a new class starting from an existing class, and globally change the names as described above, all of the definitions and references listed here will already be done for you.

Table 5-3. Summary of Xt Structure Name Conventions

Structure Name	Description
`BitmapEditClassPart`	Partial class structure typedef (usually dummy field only) used for defining `BitmapEditClassRec` in *P.h* file.
`BitmapEditClassRec`	Complete class structure typedef, declared `extern` in *P.h* file, used for initializing class structure in *.c* file.
`bitmapEditClassRec`	Name of complete class structure allocated in *.c* file, declared `extern` in *P.h* file, allocated in *.c* file, used as superclass in class record initialization of subclasses of this widget (*.c* file).
`BitmapEditPart`	Partial instance structure typedef, used for defining `BitmapEditRec` in *P.h* file.
`BitmapEditRec`	Complete instance structure typedef, also used to initialize `widget_size` field of class record in *.c* file.
`_BitmapEditRec`	Type of `BitmapEditRec`, used for defining `BitmapEditWidget` pointer in *.h* file.

Table 5-3. Summary of Xt Structure Name Conventions (continued)

Structure Name	Description
`_BitmapEditClassRec`	Type of `BitmapEditClassRec`, used for defining `BitmapEditWidgetClass` pointer in *.h* file.
`BitmapEditWidget`	Pointer to `BitmapEditRec` (complete instance structure), used to reference instance structure fields in *.c* file (passed as argument to methods).
`BitmapEditWidgetClass`	Pointer to `_BitmapEditClassRec`, used to cast `bitmapEditClassRec` for superclass in class record initialization of subclasses of this widget (*.c* file).
`bitmapEditWidgetClass`	Of type `WidgetClass`, address of `bitmapEditClassRec`, used in `XCreateWidget` calls to identify class to be created.

6

Basic Widget Methods

This chapter describes the initialize, expose, set_values, resize, query_geometry, and destroy methods. It explains when Xt calls each method, and describes in detail what should be in each of these methods. Among other things, these methods prepare for and do the drawing of graphics that appear in a widget. This chapter describes what the Toolkit adds to the graphics model provided by Xlib, but does not describe in detail how to draw using Xlib; this topic is described in Chapters 5, 6 and 7 of Volume One, Xlib Programming Manual. *This chapter also describes how to write action routines that work in the widget framework.*

In This Chapter:

The X Graphics Model Inside Widgets 170
The initialize Method 170
  Creating GCs 172
The expose Method 175
The set_values Method 178
The resize Method 181
The query_geometry Method 184
The destroy Method 187
Actions in the Widget Framework 188

6
Basic Widget Methods

This chapter describes the `initialize`, `expose`, `set_values`, `resize`, `query_geometry`, and `destroy` methods. These are the methods you need to write to make a functioning widget (although `destroy` is optional). The common thread in most of these methods is that they have a role in drawing graphics, although all but the `expose` method are also responsible for other things. We will describe all the responsibilities of these methods, but focus on the issues involving graphics because they are so important.

Three of these methods are called by Xt in response to application function calls:

- The `initialize` method is called when the application calls `XtVaCreateManagedWidget`.
- The `set_values` method is called when the application calls `XtVaSetValues`.
- The `destroy` method is called when the application calls `XtDestroyWidget`.

A fourth method, `expose`, is called in response to `Expose` events, which occur whenever part or all of a widget's window becomes newly visible on the screen.*

When the parent widget, which is usually a geometry-managing Composite or Constraint widget, needs to resize one of its children, it may call the child's `query_geometry` method to find the child's opinion of the proposed change. Once the parent has actually resized the child, Xt calls the child's `resize` method, which is responsible for calculating private instance variables, so that the child is prepared to redraw itself in its new window size.

Applications that use the Toolkit should do all their drawing in widgets. Although it is possible to create normal X windows and draw into them from the application code, you lose the advantage of Xt's event dispatching, event filtering, and other features. It's just as easy to draw in a widget, and it's easier to add processing of user input as described in Chapter 7, *Events, Translations, and Accelerators*.

*Note that `Expose` events are also handled through the translation mechanism like any other event. When they are present in the translation table, the action registered for `Expose` events will be called in addition to the `expose` method. Widgets do not normally include the `Expose` event in their translation table, because they already have the `expose` method. Applications might, conceivably, have `Expose` in their translation table to add drawing capability to a widget, although it is difficult to calculate where to do this drawing from the application since the widget's size may change.

6.1 The X Graphics Model Inside Widgets

Section 4.3.1 described the X graphics model and how drawing from the application is accomplished. In summary, the process consisted of creating GCs and then drawing from a function that Xt calls on receipt of `Expose` events. Drawing from inside a widget follows the same general procedure, but the code is organized differently.

Inside a widget, you normally create GCs with the Xt routine `XtGetGC` instead of the Xlib routine `XCreateGC`. `XtGetGC` is similar to the Xlib routine except that it arranges for the sharing of GCs among widgets within an application. This is important because each GC has some overhead and servers can achieve better performance when handling fewer GCs. Because Xt applications often have many instances of the same widget class, each needing the same GC characteristics (unless the user specifies a different color for each one), Xt arranges for them to share GCs when possible.

Xt organizes GC creation and drawing into separate methods. Setting initial values for the GC and creating the GC is done in the `initialize` method, and actual drawing is done in the `expose` method. This makes it very easy to find this code in existing widgets and straightforward to write this code for new widgets.

Since Xt allows resources to be changed during program execution by calling `XtVaSetValues`, the widget code must be prepared to change the GC at any time if the GC components depend on resource values. The `set_values` method calculates the new values based on resource changes and updates the appropriate GCs.

The next six major sections discuss the code needed to implement the `initialize`, `expose`, `set_values`, `resize`, `query_geometry`, and `destroy` methods.

6.2 The initialize Method

As described in Section 5.3.5.2, the `initialize` method has basically one function: it sets instance structure members (also called instance variables).* This job has two parts: setting the initial values of private instance variables, and checking to make sure that the values of the public instance variables are valid (since they are user configurable). Since the `initialize` method is upward chained (defined in Section 5.3.5.2), the method for this class needs to initialize only the fields present in the instance part structure for this class. The exceptions are `width` and `height`. Even though they are instance variables of Core, they need to be checked by the subclass, because only this class knows its desired initial size.

Example 6-1 shows the `initialize` method from BitmapEdit.

*As described in Chapter 11, *Geometry Management*, the `initialize` method can also be used for creating child widgets, to build a compound widget. This is a way of getting around Xt's geometry management scheme, and is not frequently done.

Example 6-1. The initialize method

```
/* ARGSUSED */
static void
Initialize(treq, tnew, args, num_args)
Widget treq, tnew;
ArgList args;
Cardinal *num_args;
{
    BitmapEditWidget new = (BitmapEditWidget) tnew;
    new->bitmapEdit.cur_x = 0;
    new->bitmapEdit.cur_y = 0;
    /*
      *  Check instance values set by resources that may be invalid.
      */

    if ((new->bitmapEdit.pixmap_width_in_cells < 1) ||
                (new->bitmapEdit.pixmap_height_in_cells < 1))  {
        XtWarning("BitmapEdit: pixmapWidth and/or pixmapHeight\
                is too small (using 10 x 10).");
        new->bitmapEdit.pixmap_width_in_cells = 10;
        new->bitmapEdit.pixmap_height_in_cells = 10;
    }

    if (new->bitmapEdit.cell_size_in_pixels < 5) {
        XtWarning("BitmapEdit: cellSize is too small (using 5).");
        new->bitmapEdit.cell_size_in_pixels = 5;
    }

    if ((new->bitmapEdit.cur_x < 0) ||  (new->bitmapEdit.cur_y < 0)) {
        XtWarning("BitmapEdit: cur_x and cur_y must be\
                non-negative (using 0, 0).");
        new->bitmapEdit.cur_x = 0;
        new->bitmapEdit.cur_y = 0;
    }

    /*
     * Allocate memory to store array of cells in bitmap, if not
     * already done by application.
     */
    if (new->bitmapEdit.cell == NULL)
        new->bitmapEdit.cell =
                XtCalloc(new->bitmapEdit.pixmap_width_in_cells
                * new->bitmapEdit.pixmap_height_in_cells, sizeof(char));

    /* Calculate useful values */
    new->bitmapEdit.pixmap_width_in_pixels =
            new->bitmapEdit.pixmap_width_in_cells
            * new->bitmapEdit.cell_size_in_pixels;

    new->bitmapEdit.pixmap_height_in_pixels =
            new->bitmapEdit.pixmap_height_in_cells
            * new->bitmapEdit.cell_size_in_pixels;

    /* set initial size of window */
    if (new->core.width == 0) {
        if (new->bitmapEdit.showAll == False)
            new->core.width = (new->bitmapEdit.pixmap_width_in_pixels >
                    DEFAULTWIDTH) ? DEFAULTWIDTH :
                    (new->bitmapEdit.pixmap_width_in_pixels);
        else
```

Example 6-1. The initialize method (continued)

```
            new->core.width = new->bitmapEdit.pixmap_width_in_pixels;
    }

    if (new->core.height == 0) {
        if (new->bitmapEdit.showAll == False)
            new->core.height = (new->bitmapEdit.pixmap_height_in_pixels >
                    DEFAULTWIDTH) ? DEFAULTWIDTH :
                    (new->bitmapEdit.pixmap_height_in_pixels);
        else
            new->core.height = new->bitmapEdit.pixmap_height_in_pixels;
    }

    CreateBigPixmap(new);

    GetDrawGC(new);
    GetUndrawGC(new);
    GetCopyGC(new);

    DrawIntoBigPixmap(new);
}
```

Even though the specific instance variables initialized here are particular to BitmapEdit, the techniques are common to all `initialize` methods. Some private instance variables, such as `cur_x` and `cur_y`, do not depend on resource settings and are simply initialized to a fixed value. Public instance variables are checked; if their values are out of range, `XtWarning` is called to print a message on the standard output and they are instead initialized to a fixed value. Some private instance variables are set based on public instance variables. GCs are the most common example.

As you should recall from Chapter 5, *Inside a Widget*, all the variables set in `initialize` are declared in the `BitmapEditPart` instance structure in the private header file.

Note that most of the arguments of `initialize` are rarely used. When `initialize` is called, the *`request`* argument is an instance structure containing the same values as *`new`*. But as *`new`* is changed, *`request`* provides a sometimes useful record of the resource settings originally made by the user. The *`args`* and *`num_args`* arguments contain the resource settings provided in the call to `XtVaCreateManagedWidget`. These are sometimes useful in distinguishing between settings made by the application (*`args`*) from settings made by the user (which already appear in *`new`* and *`request`*). As of R4, the code in the `initialize` `hook` method, described in Chapter 9, *Resource Management and Type Conversion*, can optionally be placed in the `initialize` method.

6.2.1 Creating GCs

In any widget that does drawing, some of the private instance variables will hold the IDs of GCs created in the `initialize` method. These variables will be read when the GCs are needed in the `expose` method, and will be reset if necessary in the `set_values` method. Example 6-1 called separate routines, `GetDrawGC`, `GetUndrawGC`, and `GetCopyGC` to create the GCs. Example 6-2 shows these routines.

A program prepares for creating a GC by setting the desired characteristics of the GC into members of a large structure called XGCValues (defined by Xlib), and specifying which members of XGCValues it has provided by setting a bitmask. This bitmask is made by ORing the GC mask symbols defined in *<X11/X.h>*. Each bitmask symbol represents a member of the XGCValues structure. Every GC field has a default value, so only those values that differ from the default need to be set. The default GC values are discussed in Volume One, *Xlib Programming Manual*.

Widget code normally creates a GC with the XtGetGC call, not the Xlib analogue XCreateGC. XtGetGC keeps track of requests to get GCs by all the widgets in an application, and creates new server GCs only when a widget requests a GC with different values. In other words, after the first widget creates a GC with XtGetGC, any subsequent widget that calls XtGetGC to create a GC with the same values will get the same GC, not a new one. Using this client-side caching also reduces the number of requests to the server.

Xlib provides XChangeGC to change the values in an existing GC. Xt provides no analogue. Because of the sharing of GCs allocated with XtGetGC, you must treat them as read-only. If you don't, you may confuse or break other widgets. In cases where your widget creates a GC that needs changing in such a way that it is impractical to create a new GC each time, it may be appropriate to use the Xlib call XCreateGC instead of the Xt call XtGetGC.

BitmapEdit must use XCreateGC for two of its three GCs because these GCs are for drawing into pixmaps of depth 1, while XtGetGC always creates GCs for drawing into windows or pixmaps of the default depth of the screen. In other words, XtGetGC would work on a monochrome screen, but not on color. However, this is an unusual situation. Most widgets can use XtGetGC exclusively.

Example 6-2. Creating GCs from the initialize method

```
static void
GetDrawGC(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground | GCDashOffset |
            GCDashList | GCLineStyle;

    /*
     * Setting foreground and background to 1 and 0 looks like a
     * kludge but isn't.  This GC is used for drawing
     * into a pixmap of depth one.  Real colors are applied with a
     * separate GC when the pixmap is copied into the window.
     */
    values.foreground = 1;
    values.background = 0;
    values.dashes = 1;
    values.dash_offset = 0;
    values.line_style = LineOnOffDash;

    cw->bitmapEdit.draw_gc = XCreateGC(XtDisplay(cw),
            cw->bitmapEdit.big_picture, mask, &values);
}
```

Example 6-2. Creating GCs from the initialize method (continued)

```
static void
GetUndrawGC(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground;

    /* Looks like a kludge but isn't--see comment in GetDrawGC */
    values.foreground = 0;
    values.background = 1;

    cw->bitmapEdit.undraw_gc = XCreateGC(XtDisplay(cw),
            cw->bitmapEdit.big_picture, mask, &values);
}

static void
GetCopyGC(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground;

    values.foreground = cw->bitmapEdit.foreground;
    values.background = cw->core.background_pixel;

    /* This GC is the same depth as screen */
    cw->bitmapEdit.copy_gc = XtGetGC(cw, mask, &values);
}
```

As shown in Example 6-2, some GC components are set based on other instance variables, and some are just hardcoded. In this example, the foreground pixel value (color) in the GC is set to be the value of the `foreground` instance variable, which is derived from a resource of BitmapEdit. The background pixel value is derived from a resource of the Core widget. (Core defines a background resource but no foreground.) On the other hand, the `line_style` member of `XGCValues` is set to `LineOnOffDash` regardless of resource settings. In general, you hardcode the GC components that you don't want to be user customizable. (All these instance variables are defined in the *`yourwidget`*`Part` structure defined in *yourwidgetP.h*.)

Certain GC parameters are traditionally handled as resources rather than being hardcoded. Colors and fonts are the most basic examples. In the X Window System you can't use a color until you allocate a pixel value for it, and you can't use a font until you load it. The code that processes resources called by `XtAppInitialize` automatically allocates and loads the colors and fonts specified as resources, and sets the instance variables to the right representation type as specified in the resource table. Therefore, it is actually easier to provide user customizability than to convert values to the representation types required to hardcode them.

How the resource conversion process works will be described in more detail in Chapter 9, *Resource Management and Type Conversion*. To decide which GC settings you need for your application, see Chapter 5, *The Graphics Context*, in Volume One, *Xlib Programming Manual*.

6.3 The expose Method

The `expose` method is responsible for initially drawing into a widget's window and for redrawing the window every time a part of the window becomes exposed. This redrawing is necessary because the X server does not maintain the contents of windows when they are obscured. When a window becomes visible again, it must be redrawn.

The `expose` method usually needs to modify its drawing based on the geometry of the window and other instance variables set in other methods. For example, the Label widget will left-justify, center, or right-justify its text according to the `XtNjustify` resource, and the actual position to draw the text depends on the widget's current size. The Label widget has an instance variable called `justify`, which is set initially in the `initialize` method, and is read in Label's `expose` method.

Another factor to consider when writing the `expose` method is that many widgets also draw from action routines, in response to user events. For example, BitmapEdit toggles bitmap cells in action routines. The `expose` method must be capable of redrawing the current state of the widget at any time. This means that action routines usually set instance variables when they draw so that the `expose` method can read these instance variables and draw the right thing.

Most widgets keep track of what they draw in some form of arrays or display lists. When they need to redraw, they simply replay the saved drawing commands in the original order to redraw the window. For example, BitmapEdit keeps track of the state of each bitmap cell in a character array. It could easily traverse this array and redraw each cell that is set in the array.

However, BitmapEdit does not use this strategy. In order to improve its scrolling performance, the `expose` method copies an off-screen pixmap into the window whenever redisplay is required. The actions draw into this off-screen pixmap (in addition to updating the character array), and then call the `expose` method directly to have the correct portion of the pixmap copied to the window.

The `expose` method is passed an event that contains the bounding box of the area exposed. To achieve maximum performance it copies only this area from the pixmap to the window. The BitmapEdit actions take advantage of this, too. They manufacture an artificial event containing the bounding box of the cell to be toggled, and pass it when they call `expose`. This causes the `expose` method to copy that one cell that was just updated to the window.

Example 6-3 shows the `expose` method from the BitmapEdit widget.

Example 6-3. The expose method

```
/* ARGSUSED */

static void
Redisplay(w, event)
Widget cw;
XExposeEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    register int x, y;
```

Example 6-3. The expose method (continued)

```
    unsigned int width, height;
    if (!XtIsRealized(cw))
        return;

    if (event) {  /* called from btn-event */
        x = event->x;
        y = event->y;
        width = event->width;
        height =  event->height;
    }

    else {/* called because of expose */
        x = 0;
        y = 0;
        width = cw->bitmapEdit.pixmap_width_in_pixels;
        height = cw->bitmapEdit.pixmap_height_in_pixels;
    }

    if (DefaultDepthOfScreen(XtScreen(cw)) == 1)
        XCopyArea(XtDisplay(cw), cw->bitmapEdit.big_picture,
                XtWindow(cw), cw->bitmapEdit.copy_gc, x +
                cw->bitmapEdit.cur_x, y + cw->bitmapEdit.cur_y,
                width, height, x, y);

    else
        XCopyPlane(XtDisplay(cw), cw->bitmapEdit.big_picture,
                XtWindow(cw), cw->bitmapEdit.copy_gc, x +
                cw->bitmapEdit.cur_x, y + cw->bitmapEdit.cur_y,
                width, height, x, y, 1);
}
```

Note that the `expose` method first checks to see that the widget is realized using `XtIsRealized`. This is a precaution against the unlikely event that an instance of this widget is suddenly destroyed or unrealized by an application while `Expose` events are still pending. If this did happen, drawing on the nonexistent window would cause an X protocol error.

Next, BitmapEdit's `expose` method sets the rectangle it will redraw based on the event passed in by Xt. We also call this method directly from the action that processes button presses. That action routine creates a pseudo-event to pass to `expose` to describe the area to be drawn.

If the `compress_exposure` field of the class structure is initialized to `XtExposeCompressMultiple` (or one of a few other constants, described in Chapter 8, *More Input Techniques*), as it is in BitmapEdit, Xt automatically merges the multiple `Expose` events that may occur because of a single user action into one `Expose` event. In this case, the `Expose` event contains the bounding box of the areas exposed. BitmapEdit redraws everything in this bounding box. For widgets that are very time-consuming to redraw, you might want to use the third argument of the `expose` method, which is a region. The `Region` type is opaquely defined by Xlib (internally a linked list of rectangles). The `Region` passed into `expose` describes the union of all the areas exposed by a user action. You can use this region to clip output to the exposed region, and possibly calculate which drawing primitives affect this area. Xlib provides region mathematics routines (such as `XRectInRegion`) to compare the regions in which your widget needs to draw with the

region needing redrawing. If certain areas do not require redrawing, you can skip the code that redraws them, thereby saving valuable time. However, if this calculation is complicated, its cost/benefit ratio should be examined.

Consider the arrangement of windows shown in Figure 6-1.

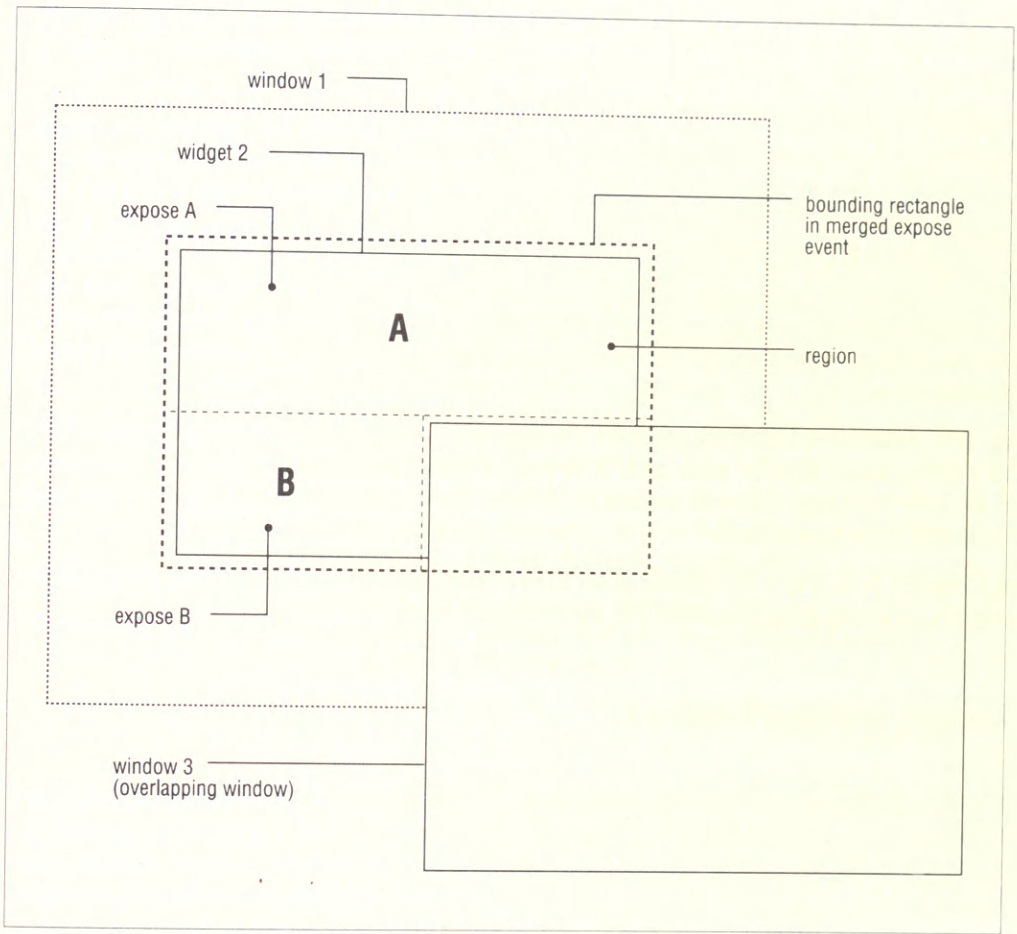


Figure 6-1. compress_exposure: 2 rectangles if XtExposeNoCompress; bounding box and region if XtExposeCompressSeries or XtExposeCompressMultiple.

Window 1 and Window 3 are other applications, and Widget 2 is an application consisting solely of our widget.

Initially, Window 3 is on top, Window 1 is behind it, and Widget 2 is hidden completely behind Window 1. When Window 1 is lowered, Widget 2 becomes visible, except where it is still overlapped by Window 3. The newly-exposed area can be described by two rectangles; A and B. If `compress_exposure` is `FALSE`, Widget 2's `expose` method will be called

twice, and passed an Expose event first describing Rectangle A, then Rectangle B. But if compress_exposure is TRUE, Widget 2's expose method will be called just once, passed an Expose event describing the bounding box of all the original Expose events (which would be the entire widget in this case), and passed a Region which is the union of the rectangles described by all of the Expose events. The region argument of the expose method is unused unless compress_exposure is TRUE. Each of these exposure-handling techniques may be the best for certain widgets. For a widget like BitmapEdit, any of the three methods will work, but the bounding box method is the most efficient and convenient. For a complete description of Expose event handling strategies, see Chapter 8, *Events*, in Volume One, *Xlib Programming Manual*.

The remainder of BitmapEdit's expose method shown in Example 6-3 consists of a single Xlib call to copy from a pixmap into the widget's window. As described in Chapter 4, *An Example Application*, BitmapEdit makes a large pixmap that is one plane deep and draws the current bitmap into it. When needed in the expose method, this pixmap just has to be copied into the window. This approach was chosen for its simplicity. When scrollbars are added, the widget is able to pan around in the large bitmap quickly and efficiently. Note that one of two Xlib routines is called based on the depth of the screen. This is because XCopyArea is slightly more efficient than XCopyPlane and should be used when running on a monochrome screen.

Note that instance variables are used for the arguments of the Xlib routines in Example 6-3. Don't worry about exactly what each Xlib routine does or the meaning of each argument. See the reference page for each routine in Volume Two, *Xlib Reference Manual*, when you need to call them in your code.

See Chapters 5, 6, and 7 in Volume One, *Xlib Programming Manual*, for more information on the GC, drawing graphics, and color, respectively.

6.4 The set_values Method

When the application calls XtVaSetValues (or XtVaSetValues) to change widget resources during run time, Xt calls the set_values method. The set_values method is where a widget responds to changes in its public instance variables. It should validate the values of the public variables, and recalculate any private variables that depend on public variables that have changed.

Example 6-4 shows the set_values method for BitmapEdit.

Example 6-4. The set_values method

```
/* ARGSUSED */
static Boolean
SetValues(current, request, new, args, num_args)
Widget current, request, new;
ArgList args;
Cardinal *num_args;
{
```

Example 6-4. The set_values method (continued)

```
    BitmapEditWidget curcw = (BitmapEditWidget) current;
    BitmapEditWidget newcw = (BitmapEditWidget) new;
    Boolean do_redisplay = False;

    if (curcw->bitmapEdit.foreground != newcw->bitmapEdit.foreground) {
        XtReleaseGC(curcw, curcw->bitmapEdit.copy_gc);
        GetCopyGC(newcw);
        do_redisplay = True;
    }

    if ((curcw->bitmapEdit.cur_x != newcw->bitmapEdit.cur_x) ||
            (curcw->bitmapEdit.cur_y != newcw->bitmapEdit.cur_y))
        do_redisplay = True;

    if (curcw->bitmapEdit.cell_size_in_pixels !=
            newcw->bitmapEdit.cell_size_in_pixels) {
        ChangeCellSize(curcw, newcw->bitmapEdit.cell_size_in_pixels);
        do_redisplay = True;
    }

    if (curcw->bitmapEdit.pixmap_width_in_cells !=
            newcw->bitmapEdit.pixmap_width_in_cells)  {
        newcw->bitmapEdit.pixmap_width_in_cells =
                curcw->bitmapEdit.pixmap_width_in_cells;
        XtWarning("BitmapEdit: pixmap_width_in_cells cannot\
                be set by XtSetValues.\n");
    }

    if (curcw->bitmapEdit.pixmap_height_in_cells !=
            newcw->bitmapEdit.pixmap_height_in_cells) {
        newcw->bitmapEdit.pixmap_height_in_cells =
                curcw->bitmapEdit.pixmap_height_in_cells;
        XtWarning("BitmapEdit: pixmap_height_in_cells cannot\
                be set by XtSetValues.\n");
    }

    return do_redisplay;
}
```

The `set_values` method is called with three copies of the widget's instance structure as arguments: *`current`*, *`request`*, and *`new`*. The *`current`* copy includes the current settings of the instance variables, and *`request`* includes the settings made through `XtVaSetValues` but not yet changed by the superclasses' `set_values` methods. The *`new`* copy is the same as *`request`* except that it has already been processed by the `set_values` methods of all superclasses.

For each public variable, `set_values` compares the current value and the new value, and if they are different, validates the *`new`* value or changes any private values that depend on it. The *`new`* copy is the only one that the method changes; *`request`* and *`current`* are only for reference. As in the `initialize` method, you have to deal only with the instance variables for your subclass, and perhaps with width and height, because the `set_values` method is downward chained. Superclass `set_values` methods take care of setting all the superclass instance fields. However, if desired, you can change superclass fields in your method since it is called last. For example, this might be useful if your class has different criteria for determining valid values or dependencies.

The *request* copy of the instance variables is used only if your class needs to decide between a superclass setting and your widget's setting; disagreements about this usually occur over the size of the widget. For more information on when to use *request*, see the reference page for XtSetValuesFunc in Volume Five, *X Toolkit Intrinsics Reference Manual*.

It is also important to notice that the set_values method, if it exists at all (that is, if it is not specified as NULL in the class structure), must return TRUE or FALSE to indicate whether the changes made to the state variables require the widget to be redrawn. If it is TRUE for this class or any superclass, Xt calls the Xlib routine XClearArea with the *exposures* argument set to TRUE (to force the background of the widget to be redrawn, which normally occurs only after Expose events), and then Xt calls the expose method. In other words, you should make set_values return TRUE whenever the changes to the instance variables will change what is drawn in the widget.

Note, however, that set_values should *not* return TRUE if width and height change, because the X server automatically generates Expose events for a window when it is resized. If you do return TRUE in this case, your expose method will be called twice.

Since the application may call XtVaSetValues before the widget is realized, it is important not to assume in the set_values method that the widget has a window. In other words, the code must use XtIsRealized to check whether the widget is realized before using the XtWindow macro, such as in a call to set window attributes.

As mentioned earlier, some of the private variables are usually GCs. Whenever the public variable for a color or font is changed through XtVaSetValues, a GC also has to be changed. This is an example of a private variable depending on a public one. GCs allocated with XtGetGC should not be changed since they may be shared by other widgets. Therefore, the normal response to a change in one of the resources on which a GC depends is to release the GC with XtReleaseGC and request a new one with XtGetGC.

If some of your widget's GCs are unlikely to be shared by other widget instances in an application (either because each instance will use different values for the GC components or there will be only one instance), and the widget needs to be able to make changes to them, and the changes are not predictable enough or few enough to reasonably create a GC for each variation, then the XtGetGC/XtReleaseGC approach is not ideal. What happens in this situation is that XtGetGC creates a new GC and XtReleaseGC frees the old GC every time a change is made, even if the changes are small. It is more efficient in this situation to change the affected values in the existing GC. But this can be done only with an Xlib routine—there is no Toolkit routine for changing a GC. To implement this approach, you create, modify, and free the unusual GC using Xlib routines only. You call XCreateGC in the initialize method to create the GC, XChangeGC in the set_values method to change the GC, and XFreeGC in the destroy method to free the GC. All of these Xlib routines are described in Chapter 5, *The Graphics Context*, in Volume One, *Xlib Programming Manual*.

XtGetGC always creates a GC that can only be used on drawables (windows and pixmaps) of the default depth of the screen. Drawing into a pixmap of depth one using a GC created with XtGetGC works on a monochrome display, but usually does not work on a color display. (The depth is the number of bits per pixel used to represent colors on the screen.) This is another situation in which you will need to call XCreateGC instead of XtGetGC.

Note also that BitmapEdit does not allow two of its resources to be changed by `XtVaSetValues`; they can be set only until the widget is created. To disallow changes to a resource, the `set_values` method must actively set the value in the new widget (`newcw`) to the value in the current widget (`curcw`), wiping out the setting made through `XtVaSetValues`. It is also advisable that the widget print a message describing that the resource cannot be set through `XtVaSetValues`.

The `XtWarning` call used in the example is simply an alternative to calling `fprintf`, that helps to make all error messages uniform, and is described in Chapter 13, *Miscellaneous Toolkit Programming Techniques*. As of R4, the code in the `set_values_hook` method, described in Chapter 9, *Resource Management and Type Conversion*, can optionally be placed in the `set_values` method.

6.5 The resize Method

When a widget is used in an application, it is created with a parent that will manage its geometry. Depending on the layout policy of this parent widget in the application, the widget's window may change size when the application is resized.* Most widgets need to recalculate the position and size of their graphics when their window changes size. This is the job of the `resize` method.

The `resize` method is passed only one argument, the widget instance structure pointer (of type `Widget`). This structure contains the new position, size, and border width of the widget's window. The method changes any instance part fields that depend on the size or position of the widget. When the `resize` method returns, Xt calls the `expose` method, regardless of whether or not the contents need redrawing. (It is a basic characteristic of the X server that it generates `Expose` events when a window is resized.)

A Label widget whose text is centered would reset the starting position of its text in the `resize` method.

In some widgets, it takes some thought to determine the correct response to resizing. Take BitmapEdit, for example. BitmapEdit can be configured to show only a portion of the bitmap, so that scrollbars can pan around in the complete bitmap. When the application is resized, should BitmapEdit show more cells or increase the cell size? Up to the point where the entire bitmap is shown, it is easier to increase the number of cells shown. When BitmapEdit is resized larger than necessary to show the entire bitmap, it should probably increase the cell size. We will use this strategy in the `resize` method of the BitmapEdit widget, which is shown in Example 6-5.†

*Also note that the application may be resized when it is first mapped on the screen, when the user sizes the rubberband outline of the application provided by most window managers. (You may not think of this as resizing, but it is.) When this happens, the application has already created its widgets and the widgets have created windows. Therefore, the `resize` method of a widget will be called if its parent widget is forced to resize it.

†This resize strategy does have one problem; it never reduces the cell size. This is not a serious problem because BitmapEdit has a resource that controls the cell size. The application that uses this widget could provide a user interface for setting the cell size if the application writer was concerned about this problem.

Example 6-5. BitmapEdit: the resize method

```
/* ARGSUSED */
static void
Resize(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    /* resize does nothing unless new size is bigger than entire pixmap */
    if ((cw->core.width > cw->bitmapEdit.pixmap_width_in_pixels) &&
            (cw->core.height > cw->bitmapEdit.pixmap_height_in_pixels)) {
        /*
         * Calculate the maximum cell size that will allow the
         * entire bitmap to be displayed.
         */
        Dimension w_temp_cell_size_in_pixels, h_temp_cell_size_in_pixels;
        Dimension new_cell_size_in_pixels;

        w_temp_cell_size_in_pixels = cw->core.width /
                cw->bitmapEdit.pixmap_width_in_cells;
        h_temp_cell_size_in_pixels = cw->core.height /
                cw->bitmapEdit.pixmap_height_in_cells;

        if (w_temp_cell_size_in_pixels < h_temp_cell_size_in_pixels)
            new_cell_size_in_pixels = w_temp_cell_size_in_pixels;
        else
            new_cell_size_in_pixels = h_temp_cell_size_in_pixels;

        /* if size change mandates a new pixmap, make one */
        if (new_cell_size_in_pixels != cw->bitmapEdit.cell_size_in_pixels)
            ChangeCellSize(cw, new_cell_size_in_pixels);
    }
}

static void
ChangeCellSize(w, new_cell_size)
Widget w;
int new_cell_size;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int x, y;

    cw->bitmapEdit.cell_size_in_pixels = new_cell_size;

    /* recalculate variables based on cell size */
    cw->bitmapEdit.pixmap_width_in_pixels =
            cw->bitmapEdit.pixmap_width_in_cells *
            cw->bitmapEdit.cell_size_in_pixels;

    cw->bitmapEdit.pixmap_height_in_pixels =
            cw->bitmapEdit.pixmap_height_in_cells *
            cw->bitmapEdit.cell_size_in_pixels;

    /* destroy old and create new pixmap of correct size */
    XFreePixmap(XtDisplay(cw), cw->bitmapEdit.big_picture);
    CreateBigPixmap(cw);

    /* draw lines into new pixmap */
    DrawIntoBigPixmap(cw);

    /* draw current cell array into pixmap */
    for (x = 0; x < cw->bitmapEdit.pixmap_width_in_cells; x++) {
```

Example 6-5. BitmapEdit: the resize method (continued)

```
        for (y = 0; y < cw->bitmapEdit.pixmap_height_in_cells; y++) {
            if (cw->bitmapEdit.cell[x + (y *
                    cw->bitmapEdit.pixmap_width_in_cells)] == DRAWN)
                DoCell(cw, x, y, cw->bitmapEdit.draw_gc);
            else
                DoCell(cw, x, y, cw->bitmapEdit.undraw_gc);
        }
    }
}

static void
DoCell(w, x, y, gc)
Widget w;
int x, y;
GC gc;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    /* otherwise, draw or undraw */
    XFillRectangle(XtDisplay(cw), cw->bitmapEdit.big_picture, gc,
            cw->bitmapEdit.cell_size_in_pixels * x + 2,
            cw->bitmapEdit.cell_size_in_pixels * y + 2,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3);
}
```

Because of the two-phase resize strategy used by BitmapEdit, and because BitmapEdit uses a pixmap in its repaint strategy, this `resize` method is more complicated than most. When the widget is resized larger that necessary to show the entire bitmap in the current `cell_size`, it destroys the current pixmap and creates a new one. Since this should not happen very often and because resizing does not require extremely fast response, the time it takes to recreate the pixmap and draw into it is acceptable.

One difficulty in writing the `resize` method is that you do not have access to the old size or position of the window. The widget instance structure passed in has already been updated with the current size and position. If you need the old information, you can cache the old size in an instance part field, set first in the `initialize` method, and again at the end of the `resize` method.

It is also important to note that the `resize` method is not allowed to request that the parent resize the widget again to get a better size. (How to suggest properly to the parent that your widget be resized is described in Chapter 11, *Geometry Management*.)

6.6 The query_geometry Method

When your widget is used in an application, its parent widget will be a composite or constraint widget that manages its size and position. Parent widgets need to know the preferred size of a widget so they can make good decisions about the size of each child. The `query_geometry` method is called when the parent is about to make a size change to some of its children but is not yet sure which ones and by how much. (How geometry management works is described in Chapter 11, *Geometry Management*. For now, we are concentrating on what you need to do to write a simple widget.)

If your widget specifies `NULL` in the class structure for the `query_geometry` method, the parent will be told that your widget's current geometry is its preferred geometry. This is often wrong information. For example, if your widget has already been resized to be one-pixel-by-one-pixel because the user has resized an application to be very small, the parent would receive the message that your widget prefers to be that small. When the application is resized to be larger again, the parent will have no information on which to base its resizing decisions. Even if your widget has no particular preference for size, it is a good idea to specify the widget's default size in `query_geometry`. Then, at least, the parent has a ballpark figure for typical sizes for your widget. The parent could at least find out that BitmapEdit is intended to be larger than a Label widget.

The `query_geometry` method is passed pointers to two copies of the `XtWidgetGeometry` structure, one containing the parent's intended size for your widget, and the other to contain your reply to the parent's suggestion. The `XtWidgetGeometry` structure is shown in Example 6-6.

Example 6-6. The XtWidgetGeometry structure

```
typedef struct {
    XtGeometryMask request_mode;
    Position x, y;
    Dimension width, height;
    Dimension border_width;
    Widget sibling;
    int stack_mode;
} XtWidgetGeometry;
```

The `request_mode` field is a mask that indicates which other fields in the structure are set. It is a bitwise OR of any or all of the symbolic constants shown in Table 6-1.

Table 6-1. XtWidgetGeometry request_mode Symbols

Symbol	Description
`CWX`	The *x* coordinate of the widget's top left corner is specified.
`CWY`	The *y* coordinate of the widget's top left corner is specified.
`CWWidth`	The widget's width is specified.
`CWHeight`	The widget's height is specified.
`CWBorderWidth`	The widget's borderwidth is specified.

Table 6-1. XtWidgetGeometry request_mode Symbols (continued)

Symbol	Description
`CWSibling`	A sibling widget is specified, relative to which this widget's stacking order should be determined.
`CWStackMode`	A `stack_mode` value is present, specifying how this widget should be stacked relative to the widget identified as *`sibling`*.

The `sibling` and `stack_mode` fields are used together to indicate where in the stacking order of its siblings your widget will be placed. The symbols for `stack_mode` are `Above Below`, `TopIf`, `BottomIf`, `Opposite`, and `XtSMDontChange`. (These symbols are used singly, not combined with OR.) Their meanings are summarized in Table 6-2.

Table 6-2. XtWidgetGeometry stack_mode Symbols

Stacking Flag	Position
`Above`	*`w`* is placed just above *`sibling`*. If no *`sibling`* is specified, *`w`* is placed at the top of the stack.
`Below`	*`w`* is placed just below *`sibling`*. If no *`sibling`* is specified, *`w`* is placed at the bottom of the stack.
`TopIf`	If *`sibling`* obscures *`w`*, then *`w`* is placed at the top of the stack. If no *`sibling`* is specified, then if any sibling obscures *`w`*, *`w`* is placed at the top of the stack.
`BottomIf`	If *`w`* obscures *`sibling`*, then *`w`* is placed at the bottom of the stack. If no *`sibling`* is specified, then if *`w`* obscures any sibling, *`w`* is placed at the bottom of the stack.
`Opposite`	If *`sibling`* occludes *`w`*, *`w`* is placed at the top of the stack. If *`w`* occludes *`sibling`*, *`w`* is placed at the bottom of the stack. If no *`sibling`* is specified, then if any sibling occludes *`w`*, *`w`* is placed at the top of the stack, or if *`w`* occludes any sibling, *`w`* is placed at the bottom of the stack.
`XtSMDontChange`	Current position in stacking order is maintained.

Note that Xt's handling of stacking order is currently incomplete, and these symbols might not be honored. By default, the most recently created widget appears on the bottom.

One more issue about the `query_geometry` method must be raised before showing an example: the method's return value. The `query_geometry` method must return one of the three `enum` values `XtGeometryYes`, `XtGeometryAlmost`, or `XtGeometryNo`. `XtGeometryResult` is an `enum` name, and it is the returned type of `query_geometry`.

- If the proposed geometry is acceptable without modification, `query_geometry` returns `XtGeometryYes`.
- If the proposed geometry is not acceptable, your widget returns `XtGeometryAlmost` and sets its suggested changes to the proposed geometry back in the reply structure.
- If the proposed geometry is the same as the current geometry, `query_geometry` returns `XtGeometryNo`. This symbol is slightly misleading—think of it as `XtGeometryNoChange` (a symbol that is not defined). The symbol is `XtGeometryNo` because all three of these symbols are used in another context within composite and constraint widgets, as is described in Chapter 11, *Geometry Management*.

Note that, in all three cases, `query_geometry` must set any fields in the reply structure that it potentially cares about, even if it is only accepting the proposed geometry.

Example 6-7 shows the `query_geometry` method from BitmapEdit. All widgets should have at least this code in the method, substituting their initial size, or their current preferred size if known. (For example, a Label widget would set its preferred size based on the width and height of the current string.)

Example 6-7. BitmapEdit: the query_geometry method

```
static XtGeometryResult QueryGeometry(w, proposed, answer)
Widget w;
XtWidgetGeometry *proposed, *answer;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    answer->request_mode = CWWidth | CWHeight;

    /* initial width and height */
    answer->width = (w->bitmapEdit.pixmap_width_in_pixels >
            DEFAULTWIDTH) ? DEFAULTWIDTH :
            w->bitmapEdit.pixmap_width_in_pixels;
    answer->height = (w->bitmapEdit.pixmap_height_in_pixels >
            DEFAULTHEIGHT) ? DEFAULTHEIGHT :
            w->bitmapEdit.pixmap_height_in_pixels;

    if (  ((proposed->request_mode & (CWWidth | CWHeight))
            == (CWWidth | CWHeight)) &&
            proposed->width == answer->width &&
            proposed->height == answer->height)
        return XtGeometryYes;
    else if (answer->width == w->core.width &&
            answer->height == w->core.height)
        return XtGeometryNo;
    else
        return XtGeometryAlmost;
}
```

6.7 The destroy Method

When a widget is destroyed by the application, its `destroy` methods are invoked in subclass to superclass order. Therefore, the `destroy` method for any given class need free only memory allocated by itself; it need not worry about memory allocated by superclasses.

Any server resources created by Xt (such as GCs requested through `XtGetGC`) should be freed in the `destroy` method. In addition, if you called any Xlib routines, such as `XCreateGC`, that allocate server- or client-side resources, be sure to free them here. BitmapEdit creates pixmaps for use in the drawing process, so it must free them. It must also free the GCs it allocated.

If this is not done, then the server resources allocated for the widget will not be freed until the application exits. This is not a fatal problem. It matters only in applications that destroy widgets and then continue running for a while before they exit, which is unusual. Example 6-8 shows the `destroy` method code from the BitmapEdit widget. It frees the pixmaps created in the `initialize` method shown in Example 6-1.

Example 6-8. The destroy method

```
static void
Destroy(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    if (cw->bitmapEdit.big_picture)
        XFreePixmap(XtDisplay(cw), cw->bitmapEdit.big_picture);

    if (cw->bitmapEdit.draw_gc)
        XFreeGC(XtDisplay(cw), cw->bitmapEdit.draw_gc);

    if (cw->bitmapEdit.undraw_gc)
        XFreeGC(XtDisplay(cw), cw->bitmapEdit.undraw_gc);

    if (cw->bitmapEdit.copy_gc)
        XtReleaseGC(cw, cw->bitmapEdit.copy_gc);

    /* This memory allocated with XtCalloc in initialize method */
    XtFree(cw->bitmapEdit.cell);
}
```

If your widget allocated memory for any of its instance variables (or other global variables) using the Toolkit routines `XtMalloc` or `XtCalloc` (which operate just like the C library but add error checking), then it should free that memory here with `XtFree`.

If your widget called `XtAddEventHandler` or `XtAddTimeOut`, then you should call `XtRemoveEventHandler` and `XtRemoveTimeOut`, respectively (these routines are described in Chapter 8, *More Input Techniques*).

6.8 Actions in the Widget Framework

Although actions, strictly speaking, are not methods, writing them is part of the process of writing a simple widget. Fortunately, you have already seen action routines added from the application. Action routines look and work the same in the widget framework as in the application, except that they use instance structure fields as data instead of the application data structure fields. Example 6-9 shows the actions of BitmapEdit.

Example 6-9. BitmapEdit: action routines

```
/*ARGUSED*/
static void
DrawCell(w, event)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    DrawPixmaps(cw->bitmapEdit.draw_gc, DRAW, cw, event);
}
/*ARGUSED*/
static void
UndrawCell(w, event)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    DrawPixmaps(cw->bitmapEdit.undraw_gc, UNDRAW, cw, event);
}
/*ARGUSED*/
static void
ToggleCell(w, event)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    static int oldx = -1, oldy = -1;
    GC gc;
    int mode;
    int newx, newy;

    /* This is strictly correct, but doesn't
     * seem to be necessary */
    if (event->type == ButtonPress) {
        newx = (cw->bitmapEdit.cur_x + ((XButtonEvent *)event)->x) /
                cw->bitmapEdit.cell_size_in_pixels;
        newy = (cw->bitmapEdit.cur_y + ((XButtonEvent *)event)->y) /
                cw->bitmapEdit.cell_size_in_pixels;
    }
    else  {
        newx = (cw->bitmapEdit.cur_x + ((XMotionEvent *)event)->x) /
```

Example 6-9. BitmapEdit: action routines (continued)

```
                cw->bitmapEdit.cell_size_in_pixels;
        newy = (cw->bitmapEdit.cur_y + ((XMotionEvent *)event)->y) /
                cw->bitmapEdit.cell_size_in_pixels;
    }

    if ((mode = cw->bitmapEdit.cell[newx + newy *
            cw->bitmapEdit.pixmap_width_in_cells]) == DRAWN) {
        gc = cw->bitmapEdit.undraw_gc;
        mode = UNDRAW;
    }
    else {
        gc = cw->bitmapEdit.draw_gc;
        mode = DRAW;
    }

    if (oldx != newx || oldy != newy) {
        oldx = newx;
        oldy = newy;
        DrawPixmaps(gc, mode, cw, event);
    }
}

static void
DrawPixmaps(gc, mode, w, event)
GC gc;
int mode;
Widget w;
XButtonEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int newx = (cw->bitmapEdit.cur_x + event->x) /
            cw->bitmapEdit.cell_size_in_pixels;
    int newy = (cw->bitmapEdit.cur_y + event->y) /
            cw->bitmapEdit.cell_size_in_pixels;
    XExposeEvent fake_event;

    /* if already done, return */
    if (cw->bitmapEdit.cell[newx + newy *
            cw->bitmapEdit.pixmap_width_in_cells] == mode)
        return;

    /* otherwise, draw or undraw */
    XFillRectangle(XtDisplay(cw), cw->bitmapEdit.big_picture, gc,
            cw->bitmapEdit.cell_size_in_pixels*newx + 2,
            cw->bitmapEdit.cell_size_in_pixels*newy + 2,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3);

    cw->bitmapEdit.cell[newx + newy *
            cw->bitmapEdit.pixmap_width_in_cells] = mode;
    info.mode = mode;
    info.newx = newx;
    info.newy = newy;

    fake_event.x = cw->bitmapEdit.cell_size_in_pixels *
            newx - cw->bitmapEdit.cur_x;
    fake_event.y = cw->bitmapEdit.cell_size_in_pixels *
            newy - cw->bitmapEdit.cur_y;
```

Example 6-9. BitmapEdit: action routines (continued)

```
    fake_event.width = cw->bitmapEdit.cell_size_in_pixels;
    fake_event.height = cw->bitmapEdit.cell_size_in_pixels;

    Redisplay(cw, &fake_event);
    XtCallCallbacks(cw, XtNcallback, &info);
}
```

Notice that as in methods, the widget instance pointer passed in is declared as type `Widget`, and then, in the first line of the action, cast to the desired type. This is necessary for ANSI C conformance.

An action routine in widget code can set and read fields in the widget instance structure passed in, while an action added from the application can access its own application data structure but not the widget's internal data structure because of the encapsulation rule (see Section 2.1.6.3). This is one of the advantages of moving this code into a widget.

You should now be ready to go back near the end of Chapter 5, *Inside a Widget*, and follow the directions to write your first widget. You will need to review parts of both Chapter 5 and this chapter as the process of actually writing a widget raises new questions in your mind.

7

Events, Translations, and Accelerators

This chapter describes the complete syntax of translation tables, and describes a mechanism called accelerators for mapping events in one widget to actions in another.

In This Chapter:

Translation Table Syntax 194
The Directive 195
Selecting the Events to Translate 195
Details in Keyboard Events 198
Details in Other Event Types 199
Modifiers 201
Physical Keys Used as Modifiers 202
Default Interpretation of the Modifier List 204
Prohibiting a Modifier 204
Requiring an Exact Match 205
Paying Attention to the Case of Keysyms 205
Event Sequences 206
Special Considerations Involving Motion Events 207
Modifiers and Event Sequences 207
Using Modifiers to Specify Button Event Sequences 208
Key Event Sequences 208
Interactions Between Translations 208
Translations in Multiple Resource Files 209
Order of Translations 209
Event Sequences Sharing Initial Events 210
Event Sequences Sharing Noninitial Events 210
Accelerators 211
Event Propagation 213
Installing Accelerators in Multiple Widgets 215
Defining the Accelerator Table in the Code 216
The display_accelerators Method 216

7
Events, Translations, and Accelerators

Events drive the application. They send a great variety of information from the server to the client and from the user to the application. More knowledge of events is necessary both to use existing widgets successfully in large applications and to write widgets.

Events are sufficiently central to X that we've devoted two chapters in this book to them. This chapter provides a closer look at translations and actions; Chapter 8, *More Input Techniques*, looks at lower-level event handlers, as well as at other sources of input. Appendix C, *Event Reference*, in Volume Five, *X Toolkit Intrinsics Reference Manual*, will also be useful when you need details about any of the event types.

The basic concept and use of translations and actions has already been described. But translation tables have a complicated syntax which can be used to do much more than the simple mappings you have seen up until now. Translation tables can detect user-interface idioms such as double- and triple-button clicks or key combinations such as Shift-Meta-M. This chapter focuses on the more advanced features of translation tables.

Next, we discuss a variation of translations called *accelerators*. Accelerators bind events that occur in one widget to actions in another. This is a flexible feature with many uses. One common use is to supply a keyboard interface to a normally pointer-driven application. By adding accelerators to the top-level window of the application, a keyboard event typed with the pointer anywhere in the application can be translated into an action in the correct widget. The name "accelerator" comes from the fact that many advanced users find it faster to use keyboard shortcuts instead of menus.

As mentioned in Chapter 3, *More Techniques for Using Widgets*, it is a very good idea to specify translation tables (and accelerator tables, too) in the application-defaults file instead of hardcoding them in the application, especially while an application is under development. This lets you develop the translations and accelerators without recompiling the application every time you want to make a change.

7.1 Translation Table Syntax

If you are reading this book in sequence, you've already seen translations used many times. However, we haven't given a formal description of their syntax or a complete listing of the events you can translate.

A translation table consists of an optional directive, which specifies how the table should be merged with any other existing translation tables, followed by a series of production rules, of the form:

[*modifier_list*] <*event*> [, <*event*> ...] [(*count*)] [*detail*] : *action* ([*arguments*]) [*action* ...]

where brackets ([ ]) indicate optional elements, an ellipsis (...) indicates repetition, and italics indicate substitution of an actual modifier, event, detail, or action name.

At a minimum, a translation must specify at least one event, specified by a predefined event name or abbreviation enclosed in angle brackets; a colon separator; and at least one action. However, a sequence of events can be specified; likewise, more than one action can be invoked as a result. The scope of event matching can be limited by one or more optional modifiers, and, in the case of some events, by a "detail" field that specifies additional information about the event. (For example, for key events the detail field specifies which key has been pressed.) Repeated occurrences of the same event (e.g., a double-click) can be specified by a count value in parentheses. A colon and optional white space separates the translation and the action.

The examples below are all valid translations:

```
<Enter>:  doit()                          invoke doit() on an EnterWindow event
<Btn1Down>,<Btn1Up>:  doit()              invoke doit() on a click of Button 1
<Btn1Up>(2):  doit()                      invoke doit() on a double-click of Button 1
Button1<Btn2Down>,<Btn2Up>:  doit()
      invoke doit() on a click of Button 2 while Button 1 is held down
Shift<BtnDown>:  doit()
      invoke doit() on a click of any button while the shift key is held down
<Key>y:  doit()                           invoke doit() when the y key is pressed
```

A translation table is a single string, even when composed of multiple translations. If a translation table consists of more than one translation, the actual newlines are escaped with a backslash (except for the last one), and character newlines are inserted with the `\n` escape sequence, as you've seen demonstrated in examples throughout this book.

The following sections provide additional detail on each of the elements of an event translation. We'll talk first about the directive, followed by event specifiers, details, modifiers, and counts. We'll also provide some pointers on the proper sequence of translations, and discuss what happens when translations overlap.

7.1.1 The Directive

As we've already seen, the three possible directives are `#replace`, `#override`, or `#augment`.

`#replace` says to completely replace the value of the `translations` resource in the widget. If no directive is specified, `#replace` is the default.

The difference between `#override` and `#augment` is more subtle. They differ in how they treat event combinations in the new translation that also appear in the old. With `#override`, the new translation takes priority, while with `#augment`, the old one does.

For example, say a widget has a default translation table that includes a translation for the Return key, and an application that uses this widget has an application-defaults file consisting of a translation table that applies to this widget. If the translation table in the application-defaults file began with `#override`, any translation for the Return key would take priority over the default translation for the Return key. On the other hand, if the translation table in the application-defaults file began with `#augment`, the default translations would take priority.

Remember that the difference between `#augment` and `#override` is only in the treatment of overlapping translations. Any translations added with either directive for events that do not already appear in the existing translation table will always be added.

Because of potential overlap between translations, when you are debugging your intended translations it is best to use `#replace` at first, then test with `#augment` or `#override` as appropriate once you are sure that the translations themselves have the desired effect.

7.1.2 Selecting the Events to Translate

An X event is a packet of data sent by the server to a client in response to user behavior or to window system changes resulting from interactions between windows. There are 33 different types of events defined by X. Most (though not all*) events can be thought of as occurring in a window: the pointer entering or leaving a window, pointer motion within a window, pointer button presses, key presses, and so on. However, most events are not sent to a window unless the window has explicitly selected that event type; events are selected on a per-window basis. In Xlib programming, events are selected by specifying an event mask as a window attribute. Some events, which are sent to all windows regardless of whether a window has selected them or not, are called non-maskable events. One of the most complex aspects of Xlib programming is designing the event loop, which must take into account all of the possible events that can occur in a window.

Xt's translation manager selects the events that are specified in the current translation table for each widget. In translations, events can be specified either by their actual names, as shown in column 1 of Table 7-1, or by means of the abbreviations shown in column 2. A

*Others reflect internal changes in the window system not related to any window. For example, a mapping `Notify` event occurs in response to a change in the mappings of keysyms (portable key symbols) to keycodes (actual codes generated by physical keys).

complete reference to each event type is provided in Appendix C, *Event Reference*, in Volume Five, *X Toolkit Intrinsics Reference Manual*.

Table 7-1. Event Type Abbreviations in Translation Tables

Event Type	Abbreviations	Description
ButtonPress	BtnDown	Any pointer button pressed
	Btn1Down	Pointer button 1 pressed
	Btn2Down	Pointer button 2 pressed
	Btn3Down	Pointer button 3 pressed
	Btn4Down	Pointer button 4 pressed
	Btn5Down	Pointer button 5 pressed
ButtonRelease	BtnUp	Any pointer button released
	Btn1Up	Pointer button 1 released
	Btn2Up	Pointer button 2 released
	Btn3Up	Pointer button 3 released
	Btn4Up	Pointer button 4 released
	Btn5Up	Pointer button 5 released
KeyPress	Key	Key pressed
	KeyDown	Key pressed
	Ctrl	KeyPress with Ctrl modifier
	Meta	KeyPress with Meta modifier
	Shift	KeyPress with Shift modifier
KeyRelease	KeyUp	Key released
MotionNotify	Motion	Pointer moved
	PtrMoved	Pointer moved
	MouseMoved	Pointer moved
	BtnMotion	Pointer moved with any button held down
	Btn1Motion	Pointer moved with button 1 held down
	Btn2Motion	Pointer moved with button 2 held down
	Btn3Motion	Pointer moved with button 3 held down
	Btn4Motion	Pointer moved with button 4 held down
	Btn5Motion	Pointer moved with button 5 held down
EnterNotify	Enter	Pointer entered window
	EnterWindow	Pointer entered window
LeaveNotify	Leave	Pointer left window
	LeaveWindow	Pointer left window
FocusIn	FocusIn	This window is now keyboard focus
FocusOut	FocusOut	This window lost keyboard focus
KeymapNotify	Keymap	Keyboard mappings changed
Expose	Expose	Part of window needs redrawing
GraphicsExpose	GrExp	Source of copy unavailable
NoExpose	NoExp	Source of copy available
ColormapNotify	Clrmap	Window's colormap changed
PropertyNotify	Prop	Property value changed
VisibilityNotify	Visible	Window has been obscured
ResizeRequest	ResReq	Redirect resize request to window manager

Table 7-1. Event Type Abbreviations in Translation Tables (continued)

Event Type	Abbreviations	Description
CirculateNotify	Circ	Stacking order modified
ConfigureNotify	Configure	Window resized or moved
DestroyNotify	Destroy	Window destroyed
GravityNotify	Grav	Window moved due to win gravity attribute
MapNotify	Map	Window mapped
CreateNotify	Create	Window created
ReparentNotify	Reparent	Window reparented
UnmapNotify	Unmap	Window unmapped
CirculateRequest	CircRec	Redirect stacking order change to window manager
ConfigureRequest	ConfigureReq	Redirect move or resize request to window manager
MapRequest	MapReq	Redirect window map request to window manager
MappingNotify	Mapping	Keyboard mapping changed
ClientMessage	Message	Client-dependent
SelectionClear	SelClr	Current owner is losing selection
SelectionNotify	Select	Selection is ready for requestor
SelectionRequest	SelReq	Request for selection to current owner

Many of these events are handled automatically by the Toolkit separately from the translation mechanism. For example, Expose events are automatically sent to the expose method of the appropriate widget. When there is also a translation for Expose events, the action is called in addition to the expose method. The only reason you would need a translation for Expose events is to add drawing to a Core widget. Because the Core widget doesn't have an expose method, there is rarely, if ever, a case where there is both an expose method and a translation for Expose events for the same widget. The point to remember is that any event can be specified in a translation, even when that event is also used in some other way by Xt.

GraphicsExpose and NoExpose events are useful when your application copies from a visible window with XCopyArea or XCopyPlane. They notify the application when part of the source of the copy is obscured. If you want GraphicsExpose and NoExpose events, you must explicitly select them by setting the graphics_exposures GC component in the GC used for the copy. Then you can provide a translation for them, or if you are writing a widget you can have them sent to your expose method by setting the compress_exposure field of the Core structure to a special value, as described in Section 8.6.2. (While *xbitmap* and BitmapEdit use XCopyArea and XCopyPlane, they copy from an off-screen pixmap that cannot be obscured, and therefore GraphicsExpose and NoExpose events are not needed. For a description of these events, see Chapter 5, *The Graphics Context*, in Volume One, *Xlib Programming Manual*.)

Several of the *Notify events are automatically handled by the Toolkit. Xt places this information in internal structures so that it can satisfy queries for widget positions and geometries without querying the server. MappingNotify events are automatically han-

dled; Xt gets the current keyboard mapping from the server. `VisibilityNotify` events are handled automatically if the `visible_interest` field in the Core structure is set to `TRUE` (see Section 8.4.1).

With the exception of `SelectionRequest`, the `*Request` events are intended for use only by window managers. The selection events are described in Chapter 10, *Interclient Communications*.

Whether or not they are already handled by Xt or in translations, events can also be selected explicitly by installing event handlers in a widget (as described in Chapter 8, *More Input Techniques*).

7.1.3 Details in Keyboard Events

As you've seen, Xt provides special event abbreviations to specify which pointer button was pressed or released. With key events, this approach would be a little impractical, since there are so many keys. Instead, the Translation Manager allows you to follow the event specification with an optional detail field. That is:

```
<Key>:  doit()
```

means that you want the `doit()` action to be invoked when any key has been pressed, while:

```
<Key>y:  doit()
```

means that you want the `doit()` action to be invoked only if the *y* key has been pressed.

What you actually specify as the detail field of a Key event is a keysym, as defined in the header file *<X11/keysymdef.h>*.* Keysyms begin with an `XK_` prefix, which is omitted when they are used in translations.

Before we explain any further, we need to review X's keyboard model, which, like many things in X, is complicated by the design goal of allowing applications to run equally well on machines with very different hardware.

The keyboard generates `KeyPress` events, and may or may not also generate `KeyRelease` events. These events contain a server-dependent code that describes the key pressed, called a *keycode*. *Modifier* keys, such as Shift and Control, generate events just like every other key. In addition, all key events also contain information about which modifier keys and pointer buttons were held down at the time the event occurred.

Xt provides a routine that translates keycode and modifier key information from an event into a portable symbolic constant called a *keysym*. A keysym represents the meaning of the key pressed, which is usually the symbol that appears on the cap of the key. For example, when the *a* key is pressed, the keysym is `XK_a`; when the same key is pressed while the Shift key is held down, the resulting keysym is `XK_A` (upper case). Note that even though both the *a* and

*Note, however, that this file includes foreign language keysym sets that are not always available. Only the MISCELLANY, LATIN1 through LATIN4, and GREEK sets are always available.

the *A* events have the same keycode, they generate a different keysym because they occurred with different modifier keys engaged. (The mapping between keycodes and keysyms is discussed further in Section 13.4.)

You may specify either the keysym name, omitting the `XK_` prefix, or its hexadecimal value (also shown in *<X11/keysymdef.h>*), prefixed by `0x` or `0X` (zero followed by *x* or *X*).

This is fairly straightforward, though there are several provisos:

- The keysym for nonalphanumeric keyboard keys may not always be obvious; you will need to look in *<X11/keysymdef.h>*. For example, the keyboard for the Sun workstation has keys labeled "Left" and "Right." Their keysyms are `XK_Meta_L` and `XK_Meta_R`, respectively. Fortunately, most common named keys have mnemonic keysym names (`XK_Return`, `XK_Linefeed`, `XK_BackSpace`, `XK_Tab`, `XK_Delete`, and so on) so you can often get by without looking them up. Notice, though, the small things that can trip you up: BackSpace, not Backspace, but Linefeed, not LineFeed. You can't count on consistency.
- The definitions in *<X11/keysymdef.h>* spell out keysym names for punctuation and other special characters. For example, you'll see `XK_question` rather than `XK_?`, and so on. Nonetheless, you need not use this long keysym name; the Translation Manager will accept the equivalent single character, as long as it is a printing character. This is also true of digits. There are a few exceptions, namely the characters that have special meaning in a translation table. If you ever need a translation for colon, comma, angle brackets, or backslash, you'll have to use the keysym name rather than the single printing character.
- The keycode-to-keysym translator built into the Translation Manager makes case distinctions only if the key event is prefaced with the colon (:) modifier. This means that `<Key>a` and `<Key>A` will be treated identically by default. We'll return to this subject when we discuss modifiers.

In sum, the following are all valid key event translations:

```
<Key>q:  quit()                  invoke quit() when q or Q is pressed
:<Key>?:  help()                 invoke help() when ? (but not /) is pressed
<Key>Return:  newline()          invoke newline() when the Return key is pressed
<Key>:  insert_char()            invoke insert_char() when any key is pressed
```

7.1.4 Details in Other Event Types

Key events are the most likely to require details, but details are also available for several other event types. They simplify your action function because they instruct Xt to call the action only when the event contains the matching detail. One type of detail, for `MotionNotify`, actually affects the selection of events.

Trivially, the detail values `Button1` through `Button5` can be supplied as details for `ButtonPress` or `ButtonRelease` events; because of the available abbreviations, though, these button detail values should not often be necessary. It is easier to say:

```
<Btn1Down>: quit()
```

than:

```
<BtnDown>Button1:  quit()
```

The `EnterNotify`, `LeaveNotify`, `FocusIn`, and `FocusOut` events can take as details any of the notify mode values shown in Table 7-2.

Table 7-2. Notify Mode Values for Enter, Leave, and Focus Events

Detail	Description
`Normal`	Event occurred as a result of a normal window crossing
`Grab`	Event occurred as a result of a grab
`Ungrab`	Event occurred when a grab was released

Normally, `EnterNotify` and `LeaveNotify` events are generated when the pointer enters or leaves a window, and `FocusIn` and `FocusOut` events are generated when a click-to-type window manager changes the keyboard focus window—the window that receives all keyboard input regardless of pointer position. In these cases the detail is `Normal`. But these events are also generated when the keyboard or pointer is grabbed by an application with a call to `XtGrabPointer`, `XtGrabButton`, `XtGrabKeyboard`, or `XtGrabKey`. A *grab*, which is usually invoked by a pop-up window, also causes keyboard and/or pointer input to be constrained to a particular window. When these events are generated because of a grab or the release of a grab, the detail is `Grab` or `Ungrab`.

Details for additional events were added in Release 4. For `MotionNotify` events you can now select normal motion events or motion hints by specifying a detail of `Normal` or `Hint`. Normal motion events are used when you need a complete record of the path of the pointer. The `compress_motion` Core field, described in Chapter 8, is for use with normal motion events. Motion hints are used when you need only periodic pointer position updates. Each time you receive a motion hint you call `XQueryPointer` (an Xlib call) to get the current pointer position. Note that the `MotionNotify` detail is the only one that affects event selection.

For `PropertyNotify`, `SelectionClear`, `SelectionRequest`, `SelectionNotify`, and `ClientMessage` events, the detail is an atom described in Table 7-3.

Table 7-3. Atom Details for Various Events

Event Type	Detail Atom
`PropertyNotify`	Property that is changing
`SelectionClear`	Selection atom (such as PRIMARY)
`SelectionRequest`	Selection atom (such as PRIMARY)
`SelectionNotify`	Selection atom (such as PRIMARY)
`ClientMessage`	Message type

For `MappingNotify`, the detail can be `Modifier`, `Keyboard`, or `Pointer`. Xt automatically does what is normally required to handle keyboard (and modifier) mapping changes. However, the *Pointer* detail might be used to make sure that the user has not disabled (by mismapping) any pointer button that is necessary for the safe operation of the application. You would simply write a translations such as `<Mapping> Pointer: ButtonMapCheck` and then call `XGetPointerMapping` in the `ButtonMapCheck` action to make sure the pointer mapping is still acceptable.

The detail values for each event type corresponds to certain members of the associated event structure, as shown in Table 7-4. See Appendix C, *Event Reference*, in Volume Five, *X Toolkit Intrinsics Reference Manual*, for more details.

Table 7-4. Event Structure Fields Used As Translation Table Hints

Event Type	Event Structure Field
`ButtonPress, ButtonRelease`	`button`
`MotionNotify`	`is_hint`
`EnterNotify, LeaveNotify`	`mode`
`FocusIn, FocusOut`	`mode`
`PropertyNotify`	`atom`
`SelectionClear`	`selection`
`SelectionRequest`	`selection`
`SelectionNotify`	`selection`
`ClientMessage`	`message_type`
`MappingNotify`	`request`

7.1.5 Modifiers

Certain events include as part of their data the state of the pointer buttons and special keyboard modifier keys at the time the event was generated. The events for which this state is available include `ButtonPress`, `ButtonRelease`, `MotionNotify`, `KeyPress`, `KeyRelease`, `EnterNotify`, and `LeaveNotify`.

For these events, you can specify a desired modifier state using one or more of the modifier keywords listed in Table 7-5. An error is generated if you specify modifiers with any other types of events.

Table 7-5. Modifiers Used in Translation Tables

Modifier	Abbreviation*	Description
Ctrl	c	Control key is held down
Shift	s	Shift key is held down
Lock	l	Caps Lock is in effect
Meta	m	Meta key is held down
Hyper	h	Hyper key is held down
Super	su	Super key is held down
Alt	a	Alt key is held down
Mod1		Mod1 key is held down
Mod2		Mod2 key is held down
Mod3		Mod3 key is held down
Mod4		Mod4 key is held down
Mod5		Mod5 key is held down
Button1		Pointer Button 1 is held down
Button2		Pointer Button 2 is held down
Button3		Pointer Button 3 is held down
Button4		Pointer Button 4 is held down
Button5		Pointer Button 5 is held down

7.1.5.1 Physical Keys Used as Modifiers

The meaning of the Meta, Hyper, Super, Alt, and Mod1 through Mod5 keywords may differ from server to server. Not every keyboard has keys with these names, and even if keysyms are defined for them in *<X11/keysymdef.h>*, they may not be available on the physical keyboard.†

For example, the file *<X11/keysymdef.h>* includes the following definitions:

```
/* Modifiers */

#define XK_Shift_L              0xFFE1  /* Left shift */
#define XK_Shift_R              0xFFE2  /* Right shift */
#define XK_Control_L            0xFFE3  /* Left control */
#define XK_Control_R            0xFFE4  /* Right control */
#define XK_Caps_Lock            0xFFE5  /* Caps lock */
#define XK_Shift_Lock           0xFFE6  /* Shift lock */

#define XK_Meta_L               0xFFE7  /* Left meta */
#define XK_Meta_R               0xFFE8  /* Right meta */
#define XK_Alt_L                0xFFE9  /* Left alt */
```

*The abbreviations did not work in MIT's R3 sample implementation, but they work in R4.

†The contents of *<X11/keysymdef.h>* are the same on every machine. What is different is the default mapping of keysyms to physical keycodes, which occurs in the server source. The only way to find out the mappings is through documentation (which is usually not available) or experimentation (as described in Chapter 11 of Volume Three, *X Window System User's Guide*, Second or Third Edition).

```
#define XK_Alt_R                0xFFEA  /* Right alt */
#define XK_Super_L              0xFFEB  /* Left super */
#define XK_Super_R              0xFFEC  /* Right super */
#define XK_Hyper_L              0xFFED  /* Left hyper */
#define XK_Hyper_R              0xFFEE  /* Right hyper */
```

There are two things you must learn before you can use these modifiers:

1. Which physical key generates a given keysym, if it is not obvious from the name.
2. Which modifiers are valid on your server.

The best way to find out what keysym a key generates is with the *xev* client, which prints detailed information about every event happening in its window. To find out a keysym, run *xev*, move the pointer into its window, and then press the key in question.

On the Sun-3, there is only one control key, on the left side of the keyboard. It generates the keysym `XK_Control_L`. There are two shift keys, one on each side, which generate the keysyms `XK_Shift_L` and `XK_Shift_R`. The Meta modifier is mapped to the keys labeled "Left" and "Right." Even though there is an Alternate keyboard key, which generates the keysym `XK_Alt_R`, this key is not recognized as a modifier. There are no Super and Hyper keys.

The list of valid modifiers can be displayed with the *xmodmap* client, as follows:

```
isla% xmodmap
xmodmap:  up to 2 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x6a),  Shift_R (0x75)
lock        Caps_Lock (0x7e)
control     Control_L (0x53)
mod1        Meta_L (0x7f),  Meta_R (0x81)
mod2
mod3
mod4
mod5
```

That is, either of the two shift keys will be recognized as the Shift modifier, the Caps Lock key as the Lock modifier, and either the Left or Right keys as the Meta modifier. The Left and Right keys are also mapped to the Mod1 modifier. The Alt key is not recognized as a modifier.

xmodmap also allows you to add keysyms to be recognized as the given modifier. For example, the following command would cause the F1 function key to be recognized as mod2:

```
isla% xmodmap -e 'add mod2 = F1'
```

For more information on *xmodmap* and *xev*, see Chapter 11 of Volume Three, *X Window System User's Guide* (Second Edition).

7.1.5.2 Default Interpretation of the Modifier List

If no modifiers are specified in a translation, the state of the modifier keys and pointer buttons makes no difference to a translation. For example, the translation:

```
<Key>q:     quit()
```

will cause the `quit` action to be invoked when the *q* key is pressed, regardless of whether the Shift, Ctrl, Meta, or Lock key is also held, and regardless of the state of the pointer buttons.

Likewise, if a modifier is specified, there is nothing to prohibit other modifiers from being present as well. For example, the translation:

```
Shift<Key>q:     quit()
```

will take effect even if the Ctrl key is held down at the same time as the Shift key (and the *q* key).

There are a number of special modifier symbols that can be used to change this forgiving state of affairs. These symbols are shown in Table 7-6.

Table 7-6. Modifier Symbols

Symbol	Description
`None`	No modifiers may be present
!	No modifiers except those explicitly specified may be present
:	Apply shift (and lock) modifier to key event before comparing
~	The modifier immediately following *cannot* be present

The syntax of these special modifiers symbols is somewhat inconsistent, and made clearest by example.

7.1.5.3 Prohibiting a Modifier

The tilde (~) is used to negate a modifier. It says that the specified modifier may not be present. For example:

```
Button1<Key>:  doit()
```

says that `doit()` should be invoked by a press of any key when button 1 is being held down, while:

```
~Button1<Key>:  doit()
```

says that `doit()` should be invoked by a press of any key *except* when button 1 is being held down.

A ~ applies only to the modifier that immediately follows it. For example:

```
~Shift Ctrl<Key>:  doit()
```

says that `doit()` should be invoked when Ctrl is held down in conjunction with any key, except if Shift is depressed at the same time.

7.1.5.4 Requiring an Exact Match

An exclamation point at the start of a modifier list states that only the modifiers in that list may be present, and must match the list exactly. For example, if the translation is specified as:

```
!Shift<Key>q:  quit()
```

the translation will take effect only if the Shift key is the only modifier present; if Caps Lock were in effect, or if a pointer button were depressed at the same time, the translation would no longer work.

The modifier `None` is the same as ! with no modifiers specified. That is:

```
None<Key>q:  quit()
```

or:

```
!<Key>q:  quit()
```

will invoke `quit` only if no modifier keys at all are pressed.

7.1.5.5 Paying Attention to the Case of Keysyms

The : modifier, like !, goes at the beginning of the modifier list and affects the entire list. This one is really in a category by itself, since it applies only to Key events.

Normally, the translations:

```
<Key>a:  doit()
```

and:

```
<Key>A:  doit()
```

have identical results: both will match either a lower-case or an upper-case *A*. Preceding the translation with a colon makes the case of the keysym significant. For example, to create commands like those in the UNIX *vi* editor, you might specify the following translations:

```
:<Key>a:  append()  \n\
:<Key>A:  appendToEndOfLine()
```

In this case, *a* and *A* are distinct. You could achieve somewhat the same result by specifying:

```
~Shift<Key>a:  append()  \n\
Shift<Key>a:  appendToEndOfLine()
```

However, this second example is both more complex and less effective. While the colon syntax will match an upper-case character generated as a result of either the Shift or Lock modifiers, the example shown immediately above will handle only Shift. You could use the following translation to prohibit both Shift and Lock from being asserted, but there is no way to specify that either Shift or Lock may be present:

```
~Shift ~Lock<Key>a:  append()  \n\
Shift<Key>a:  appendToEndOfLine()
```

Note that you cannot specify both the Shift or Lock modifier and an upper-case keysym. For example:

```
Shift<Key>A:  doit()
```

will have no effect, since you cannot further shift an uppercase character. The sequence !: at the beginning of a translation means that the listed modifiers must be in the correct state and that no other modifiers except the Shift and Lock keys may be asserted. In other words, !: by itself means the translation is triggered only if the specified key is pressed, considering case, and no other modifier is pressed.

For more details on how the Toolkit handles the ins and outs of case conversion, see Chapter 13, *Miscellaneous Toolkit Programming Techniques*. A more rigorous discussion of how the colon modifier works is also given in Appendix F, *Translation Table Syntax*, in Volume Five, *X Toolkit Intrinsics Reference Manual*.

7.1.6 Event Sequences

The left-hand side of a translation may specify a single event or a sequence of events that must occur for the action to be invoked. For example, we might want a certain action to be invoked only if the first button is clicked twice in succession.

Each event specified in the sequence is one of the abbreviations for an event type enclosed in angle brackets, optionally led by any set of modifiers and special characters, as described in Section 7.1.5. Each modifier-list/event pair in the sequence is separated by a comma.

For example, the translation to perform an action in response to a button press followed by a release of the same button (a button click) is the following:

```
<BtnDown>,<BtnUp> : doit()
```

Button press events may be specified in the translation table followed by a count in parentheses to distinguish a multiple click. The translation to detect a double click is:

```
<Btn1Down>(2) : doit()
```

or:

```
<Btn1Up>(2) : doit()
```

Notice that though they have the same effect for the user, these two translations are actually interpreted differently. The first is equivalent to saying:

```
<Btn1Down>,<Btn1Up>,<Btn1Down>:  doit()
```

while the second is equivalent to:

```
<Btn1Down>,<Btn1Up>,<Btn1Down>,<Btn1Up>:  doit()
```

A plus (+) may appear immediately after the count inside the parentheses to indicate that any number of clicks greater than the specified count should trigger the action.

The following translation detects two or more clicks:

```
<Btn1Down>(2+) : doit()
```

The maximum count that can be specified is 9.

In R3, the above click translations would be triggered even if you clicked once, went to lunch, and then clicked again when you returned. Xt uses a timing technique to solve this problem beginning in R4. The time is controlled by the `XtNmultiClickTime` application resource, which defaults to 200 milliseconds. If 200 milliseconds or more of idle time passes between clicks, Xt does not consider that a match. You can set this resource to different values for each display using `XtSetMultiClickTime`.

7.1.6.1 Special Considerations Involving Motion Events

Beware of interaction between pointer motion events and double clicking. If no motion events are specified in a translation table, these events are never selected, so there is no problem if they occur between other events. This allows a double click to be detected even if the user inadvertently jiggled the pointer in the course of the click. However, if motion events are selected for anywhere in the table, they may interfere with the expansion of events specified with the repeat notation. By definition, multiple clicks are detected only if there are no intervening events. Motion events would usually occur between the button presses and releases.

Multiple motion events will match any single motion selected event in the translation. That is:

```
<Motion>:  doit()
```

will cause `doit()` to be invoked many times, once for each motion event generated by the pointer movement, unless the `compress_motion` event filter is turned on in the Core class structure. (This filter is described in Section 8.6.2.)

7.1.6.2 Modifiers and Event Sequences

A modifier list at the start of an event sequence applies to all events in the sequence. That is:

```
Shift<BtnDown>,<BtnUp>:  doit()
```

is equivalent to:

```
Shift<BtnDown>,Shift<BtnUp>:  doit()
```

However, if modifiers and events are interspersed, the modifier applies only to the event immediately following. As an extreme case to demonstrate this behavior, consider the following translation:

```
Ctrl<Btn2Down>,~Ctrl<Btn2Up>: doit()
```

which requires that the CTRL key be depressed when pointer button 2 is pressed; however, the key must not be depressed when the button is released.

7.1.6.3 Using Modifiers to Specify Button Event Sequences

Remember that there are modifiers for specifying the state of pointer buttons as well as modifier keys. These modifiers provide another way of expressing some event sequences. For example, the following translation would invoke the action when the F1 key is pressed while button 1 is being held down:

```
Button1<Key>F1:     doit()
```

This is equivalent to:

```
<Btn1Down>,<Key>F1:  doit()
```

The following translations could be used to call an action when both the first and second pointer buttons are down, regardless of the order in which they are pressed.

```
!Button1<Button2>:doC() \n\
!Button2<Button1>:doC()
```

7.1.6.4 Key Event Sequences

The Translation Manager specification provides a special syntax for specifying a sequence of Key events—as a string. That is:

```
"yes":  doit()
```

is theoretically equivalent to:

```
<Key>y,<Key>e,<Key>s:  doit()
```

We say "theoretically" because, at least in the R3 and R4 sample implementations from MIT, the "yes" syntax does not work. It may or may not work in other implementations.

Within a sequence of key events expressed as a string (on systems where the "yes" syntax does work, if any), the special character ^ (circumflex) means Ctrl and $ means Meta. These two special characters, as well as double quotes and backslashes, must be escaped with a backslash if they are to appear literally in one of these strings.

7.1.7 Interactions Between Translations

One of the most difficult things to learn about translations is how to predict the interactions between overlapping translations.

7.1.7.1 Translations in Multiple Resource Files

Much more will be said about resource files and resource matching in Chapter 9, *Resource Management and Type Conversion*. However, in is important to make one point here.

If several resource files contain translation tables for the same widget, only the one that takes priority according to file precedence and resource matching will have any effect even if all of them specify `#augment` or `#override`. This is because the translation table is the value of the resource, and only one resource value will override all other values.

In other words, even if all the translation tables specify `#augment` or `#override`, a maximum of two translation tables are merged by Xt: the widget's default translation table and the one translation table that takes priority as determined by the process of reading and merging all the resource files (and the command line) and then the process of resource matching. If you call `XtOverrideTranslations` or `XtAugmentTranslations` in the application, this is a third level of merging, but it is unrelated to the resource files.

7.1.7.2 Order of Translations

The order of translations in a table is important, because the table is searched from the top when each event arrives, and the first match is taken. Therefore, more specific events must appear *before* more general ones. Otherwise, the more specific ones will never be reached.

For example, consider a text entry widget that wants to close itself and dispatch a callback once the user presses the Return key, indicating that entry is complete. The widget's translations might be:

```
<Key>Return:  gotData()     \n\
<Key>:        insertChar()
```

If the translations were specified in the reverse order:

```
<Key>:        insertChar() \n\
<Key>Return:  gotData()
```

`insertChar()` would be called in response to a press of the Return key, and `gotData()` would never be called.

Keeping track of translation order is made more complicated when translations are merged. For example, if one of the above lines is in the default translations of a widget class, and the other is in a translation table in a resource file, in what order are the resulting translations? Though not documented in the Xt specification, it appears that the MIT implementation of Xt places the more specific translation first.

7.1.7.3 Event Sequences Sharing Initial Events

An exception to this rule of more specific translations being placed earlier in the translation table occurs with event sequences. One translation can contain an event or event sequence that appears in another translation. If the leading event or event sequence is in common, both will operate correctly. For example, the translations:

```
<Btn1Down>,<Btn1Up>:     actionB()\n\
<Btn1Down>:              actionA()
```

execute `actionA()` when button 1 is pressed and `actionB()` when it is then released. This is true regardless of which translation appears first in the table.

This is the reason, alluded to in Chapter 2, *Introduction to the X Toolkit*, why it is not possible to bind two different actions to a single and a double click of the same button in the same widget. Specifying:

```
<Btn1Up>(2):  quit() \n\
<Btn1Down>,<Btn1Up>:  confirm()
```

will invoke the `confirm()` action on a single click and both the `confirm()` and `quit()` actions on a double click.

This behavior was a design decision on the part of the X Consortium. Otherwise, the Intrinsics could never dispatch the single-click action until the double-click interval had passed, since it would have no way of knowing if a second click was coming. Applications needing to have single and double clicks in the same widget must do so by designing the two actions appropriately, rather than relying on the Translation Manager to make the distinction.

7.1.7.4 Event Sequences Sharing Noninitial Events

If a noninitial event sequence is common between two translations, each translation will match only event sequences where the preceding event does not match the other translation. Consider the translation:

```
<Btn1Down>,<Btn1Up>:     actionB()\n\
<Btn1Up>:                actionA()
```

When a `Btn1Up` event occurs, it triggers `actionA()` only if not preceded by `Btn1Down`.

The way this works is that Xt keeps a history of recent events. When each event arrives, Xt compares this event and its immediate history to the event sequences in a translation table. An event in the sequence may have already triggered an action, but that is irrelevent. Any translation whose final event in the event sequence matches the current event and whose earlier event sequence matches the event history is matched. If there are two translations that can match a particular event sequence, then the translations are considered overlapping, the latter will override the former, and Xt will print a run-time warning message at startup (not waiting for the overlapping actions to be invoked). This is an exception to the rule stated earlier that the translation manager executes the first match.

7.2 Accelerators

Accelerators are simply translation tables registered to map event sequences in one widget to actions in another.* This is a general mechanism, but its name is based on a common use—adding a keyboard interface for menus and Command widgets—because for advanced users a keyboard interface is often faster.

Every widget has an `XtNaccelerators` resource, inherited from the Core widget class, so the actual event-action pairs are specified through the resource mechanism, just like translations. But before the `XtNaccelerators` resource can be used, the application must call `XtInstallAccelerators` or `XtInstallAllAccelerators` to specify which widget will be used as the source of the actions to be invoked, and which will be the source of the events that invoke them.

The MIT Intrinsics documentation refers to these two widgets as the *source* and *destination*. This can be confusing, especially since the arguments are referenced in reverse order in the call to `XtInstallAccelerators`:

```
void XtInstallAccelerators(destination, source)
     Widget destination;
     Widget source;
```

Just remember that the source is the widget whose actions you want executed, and the destination is the widget you want the events to be gathered in. To understand the use of the phrase *install accelerators*, think of the accelerators as the `XtNaccelerators` resource of the *source* widget, together with the actions of that widget, and the *destination* as the widget to which these actions are transplanted (i.e., "installed").

`XtInstallAccelerators` is always called from the application. (Widgets never install accelerators, because by definition they don't know about any other widgets.) In applications, accelerators can be installed any time, before or after the widgets are realized. Just before the `XtAppMainLoop` call is a good place.

As an example, we'll add accelerators to the *xbox1* application from Chapter 3, *More Techniques for Using Widgets*. This application displays two Command widgets in a Box, as shown in Figure 7-1.

*Accelerators made their debut in Release 3.

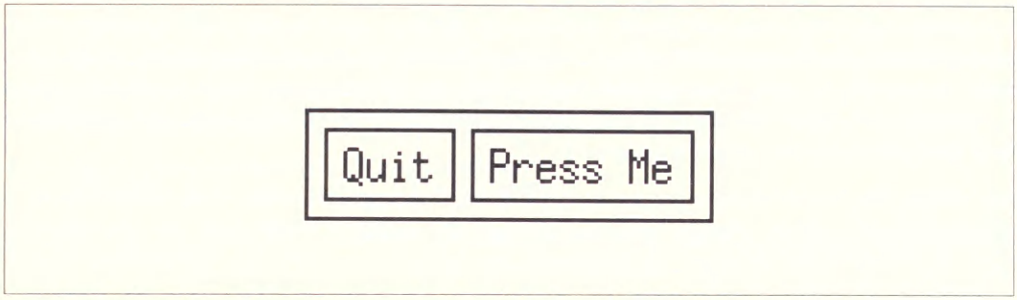


Figure 7-1. xbox1: two Command widgets in a Box

To add accelerators requires a single line of code—a call to `XtInstallAccelerators`, as shown in Example 7-1.

Example 7-1. Installing accelerators in an application

```
main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, box, quit;
      .
      .
      .
    /* allow quit widget's actions to be invoked by events in box. */
    XtInstallAccelerators(box, quit);

    XtAppMainLoop(app_context);
}
```

The actual value of the accelerator table is set through the resource mechanism. The `XtNaccelerators` resource is set for the widget whose actions are to be invoked from another widget. Xt stores this accelerator table, in compiled form, in the widget's instance structure when the widget is created. When the application calls `XtInstallAccelerators`, the accelerator table stored in the widget is merged with the translation table of the widget that will be capturing the events. (If `XtRemoveAccelerators` is called, the original translation table, prior to merging with the accelerators from the source widget, is restored.)

Example 7-2 shows a possible application-defaults file.

Example 7-2. Specifying the XtNaccelerators resource from the application-defaults file

```
*quit.label:  Quit
*pressme.label:  Press me
*quit.accelerators:\n\
        <KeyPress>q:     set() notify()
```

This resource setting will allow a *q* typed anywhere in the Box widget that comprises the *xbox1* application window to invoke the quit widget's `notify()` action (which, as you may

recall from Section 2.4, in turn invokes the `Quit` callback installed by the application). Notice that the accelerators are specified as a resource of the source widget, even though the events that invoke them will come from the destination widget.

There are some differences in the translation tables that can be used for accelerators.

- Abbreviations for event types are not valid. Only the event names that were shown in column 1 of Table 7-1 may be used (although `Key` works in place of `KeyPress`).
- The None, Meta, Hyper, Super, Alt, or ANY modifier symbols are not valid.
- The default directive for accelerator tables is `#augment` rather than `#replace`. If specified, the `#replace` directive is ignored. It is important to realize that accelerators are merged with the source widget's translations. `#override` thus says that the accelerators take priority over any matching translations in the source widget, and `#augment` says that the accelerators have lower priority.
- Parameters passed to actions must be quoted with double quotes.

7.2.1 Event Propagation

If you install accelerators in the Box widget as shown in Example 7-1, you will notice that even though the destination widget is Box, keyboard input will also be accepted in the children of `box`, including not only the quit widget but also the `pressme` widget. This is because of a characteristic of X called *event propagation*.

The server sends to the client only the types of events the client selects. Internally, Xt selects events using the Xlib call `XSelectInput`. From a Toolkit application, this is hidden because the translation mechanism automatically selects the appropriate event types specified in the translation table. (As we will see in Chapter 8, *More Input Techniques*, Xt provides a low-level event-handling mechanism that exposes event selection more directly.)

Event selection reduces the amount of network traffic, the number of events the server has to generate, and the number of unneeded events the client must process only to throw away.

Most event types are selected on a per-window basis. Each selected event that arrives at the client contains the window ID of the window it was selected for. (Non-maskable events, the last five event types in Table 7-1, are the exception; they are selected using a different mechanism or are not selected at all.)

Pointer and keyboard events propagate *through* windows that have not selected them, as shown below in Figure 7-2. That is, if the pressme widget's window has not selected `KeyPress` events, any key events that occur in that window will be passed on to its parent, the Box widget. If the Box widget's window didn't select them either, they would be passed on to the window created by the top-level Shell widget, and so on. Events propagate through the window hierarchy all the way to the root window, until they reach a window that selected them. (If no window has selected them, they are never sent by the server.) Xt treats an event that originally occurred in some descendent widget just as if it actually happened in the widget that selected the event.

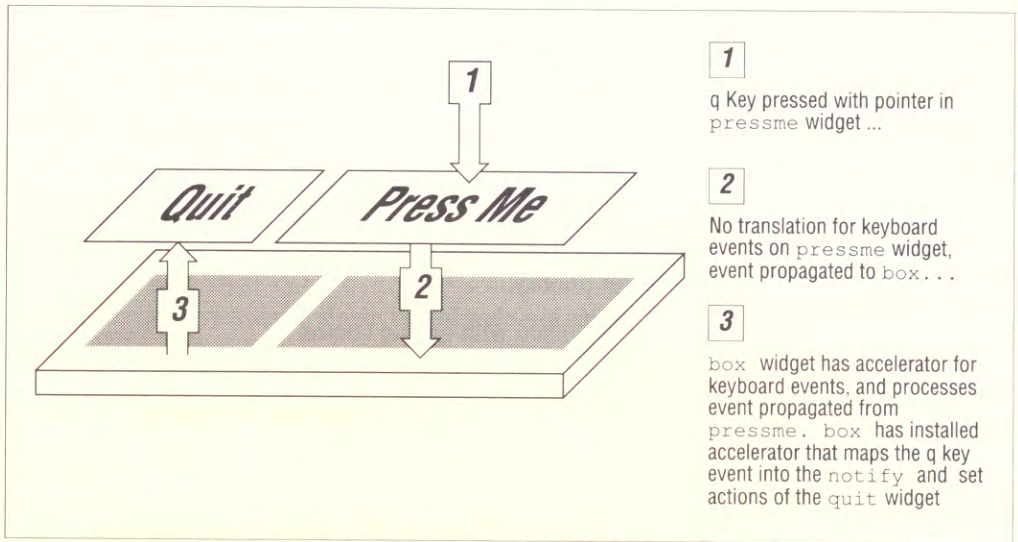


Figure 7-2. Key event propagation in xbox

Propagation is a characteristic of `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `MotionNotify`, `EnterNotify`, and `LeaveNotify` events only. As mentioned earlier, keyboard events are the events most often used as accelerators.

In the version of *xbox* we just looked at, keyboard events propagate through the Command widgets to the Box widget, because they are not selected by the Command widgets. When they reach the Box widget, they match the events specified in the accelerator table and invoke actions from the `quit` widget.

The Command widgets have a default translation table that includes pointer button events and enter/leave events. Xt, therefore, selects these events for each Command widget. Before accelerators are installed, the Box widget has no translation table and therefore has no events selected for it. When the accelerators are installed, the accelerator table is merged with the (nonexistent) translation table of the Box widget and becomes the Box's translation table. Xt then selects keyboard events for the Box widget.

But consider the resource settings in Example 7-3.

Example 7-3. Conflicting translations and accelerators

```
*quit.label:  Quit
*pressme.label:  Press me
*pressme.translations:  #override\n\
        <KeyPress>p:    set() notify()
*quit.accelerators:\n\
        <KeyPress>q:    set() notify()
```

No key event will propagate through a widget that has a translation for any key event. With the resource settings in Example 7-3, since `pressme` has a translation for the *p* key, the *q* key will not propagate through the widget. Therefore, it is often better to specify all key events as accelerators and install them on a common ancestor.

Accelerators are just merged into translations—they are not a completely different mechanism. The `#augment` (default) or `#override` directives used in accelerator tables specify whether the accelerator should override or augment existing translations for the destination widget. For example, if `box` had a translation of its own that matched the event sequence in the accelerator installed on the same widget, whether the translation or the accelerator would be invoked depends on whether `#augment` (the default) or `#override` had been specified as the accelerator directive. (This example is a little farfetched, since the Box widget defines no actions or translations—but it illustrates the point.)

When using or writing widgets, event propagation is usually important only for accelerators, because widgets are rarely layered in such a way that any of the ones that accept input are obscured.

7.2.2 Installing Accelerators in Multiple Widgets

If you want to install accelerators from more than one widget, you can call `XtInstallAccelerators` once for every widget whose actions you want executable from the destination. Alternatively, you can call `XtInstallAllAccelerators` just once for a whole application, specifying the application's top-level window as both the destination and the source. `XtInstallAllAccelerators` recursively traverses the widget tree rooted at the source, and installs the accelerators of each widget onto the destination.

For example, to install accelerators from both the quit and pressme widgets onto `box`, you could replace the call to `XtInstallAccelerators` shown in Example 7-3 with:*

Example 7-4. Installing accelerators from both command widgets

```
XtInstallAllAccelerators(box, box);
```

You could then define an application-defaults file such as the following:

Example 7-5. Accelerators resource settings for two widgets

```
*quit.label:  Quit
*pressme.label:  Press me
*pressme.accelerators:  \n\
        <KeyPress>p:     set() notify()
```

*In a more complex application, where the Box widget was not the main window but only a subarea containing command buttons, you might instead use the call:

```
XtInstallAllAccelerators(topLevel, box);
```

which would make the actions from only the widgets contained in `box` available from anywhere in the application.

Example 7-5. Accelerators resource settings for two widgets (continued)

```
*quit.accelerators:  \n\
        <KeyPress>q:     set() notify()
```

Because you would be most likely to specify distinct `KeyPress` events to invoke the actions in each Command widget, you would normally specify each accelerator resource separately, as shown above. But in the unlikely case that you wanted actions of multiple Command widgets to be invoked by the exact same event, you could do this instead. For example:

```
*Command.accelerators:  \n\
        <KeyPress>:  set() notify()
```

would invoke the `set` and `notify` actions of every Command widget, but not of widgets belonging to other classes.

Or assuming that there were multiple Box widgets in an application, one containing options, and the other commands, you might specify something like this:

```
*optionsbox*accelerators:  \n\
        <KeyPress>:  set() notify()
```

to invoke the actions of every Command widget in `optionsbox` only.

The point is that, like any resource setting, an accelerator table can apply to only one widget, to a group of children of a widget, or to an entire class, depending on how you identify the widgets for which you are setting the `XtNaccelerators` resource.

7.2.3 Defining the Accelerator Table in the Code

If you want to install a default accelerator table from within a program, you must follow similar steps to those used for translations, but with the following differences:

- You must set the `XtNaccelerators` resource instead of `XtNtranslations`.
- Instead of using `XtParseTranslationTable` to convert the ASCII translation table into Xt's internal form, use `XtParseAcceleratorTable`.
- There are no accelerator equivalents of `XtOverrideTranslations` or `XtAugmentTranslations`. Instead you can use `XtVaSetValues` and use the accelerator directive by specifying `#augment` or `#override`.

7.2.4 The display_accelerators Method

Xt calls the source widget's `display_accelerators` method when the accelerators are installed. The purpose of this method is to display to the user the installed accelerators. The method is passed a string representation of the accelerator table, which the method can theoretically manipulate into a form understandable by the user.

All of the Athena widgets inherit the Core widget's `display_accelerators` method by initializing the appropriate member of the class structure to `XtInheritDisplay-Accelerators`. This default method, as implemented in the MIT distribution, does nothing.

8

More Input Techniques

This chapter describes how to handle events with event handlers, and how to use information from the event structure inside an event handler or action routine. It also describes how to get file, pipe or socket input, how to use timeouts to call a function after a delay or at particular intervals, and how to use work procedures to do background processing. Finally, it discusses some low-level features of Xt for directly interacting with the event queue.

In This Chapter:

Event Handlers 222
- Adding Event Handlers 224
- Adding Nonmaskable Event Handlers 225
- Removing Event Handlers 226
- Adding Raw Event Handlers 227

Writing Routines That Use Specific Event Data 227
- Event Types and Structure Names 229

File, Pipe, and Socket Input 230
- Getting File Input 231
- Getting Pipe Input 233

Timeouts 234
- Visibility Interest 237

Work Procedures 239

Low-level Management of the Event Queue 241
- XtPending and XtPeekEvent 241
- Event Filters 242
- Input Sensitivity 243

8
More Input Techniques

In addition to translations, there is a lower-level mechanism called *event handlers*. Event handlers can be used from application or widget code. An event handler is a function registered with `XtAddEventHandler` to be called in response to a particular event or group of events (but not an event sequence or detail, as is possible with translations). This is a low-level, non-user-configurable means of getting events. Event handlers provide no additional capabilities over translations and actions. But they allow Xt to skip the step of searching a translation table before dispatching the event to an event handler. This speed advantage is possibly significant for events that come in large numbers, such as pointer motion events, or in applications with large translation tables. (Despite this speed difference, good performance in tracking pointer motion can be achieved with translations.)

Both event handlers and actions are passed as an argument the actual event structure that caused them to be invoked. Many event handlers and actions use data in the event. For example, a routine handling motion events may want to obtain the current pointer position from the event structure. After describing how to add and remove event handlers, we discuss the event structures and show a routine that uses specific event data.

Next, this chapter discusses three ways to register functions Xt will call for input other than events, since some programs do not live by X events alone.

- `XtAppAddInput` registers a procedure to be called when input is available from a file (or pipe or socket). This procedure can be used to perform code necessary when reads, writes, or exceptions are detected.
- `XtAppAddTimeOut` registers a function to be called at a particular relative time. This function can be used to implement delays or to execute code repeatedly at known intervals.
- `XtAppAddWorkProc` registers a work procedure. Work procedures are called when input from other sources is *not* available. They perform background tasks while the application is waiting for user input.

Finally, we provide more background on how the Toolkit dispatches events from `XtAppMainLoop`, and describe the low-level routines that can be used to construct your own main loop. We also describe Xt's event filters, which are controlled by flags in the Core class structure. These filters tell Xt whether or not to compress multiple motion, enter, leave, or expose events occurring in the same widget.

8.1 Event Handlers

An event handler is a function you provide to handle a particular type of event or group of event types. You register this function with a call to `XtAddEventHandler` or `XtInsertEventHandler`, specifying the widget in which the events are to be monitored. The difference between these two functions is described below.

You can later stop the function from being called with `XtRemoveEventHandler`. On any widget, you can register as many event handlers as you want, each for the same or for different types of events. When more than one routine is registered for an event, the order in which they are invoked is undefined.

Event handlers are infrequently used in application or widget code. None of the Athena widgets use them, and, out of the 40 applications in MIT's core distribution, only *xterm* and *xman* use them. As mentioned earlier, event handlers are useful for handling high-volume events, such as `MotionNotify` events, with maximum speed. An event handler for motion events would probably be the best way to implement a drawing program. However, translation tables provide good speed even for motion events on most systems.

Event handlers would also be useful when the type of events being handled changes frequently, because there is some overhead involved in compiling and merging translation tables. Event handlers can also be used to handle events for which the user-configurability of translations is not needed or wanted, such as `EnterWindow`, `LeaveWindow`, `FocusIn`, and `FocusOut` events.

Another possible use of event handlers is to speed the handling of certain events when there is a very large translation table. For example, editors typically have a large number of translations for various key combinations. If an editor was to accept pointer motion as well (to allow drawing of simple graphics), it might pay to handle motion events in an event handler instead of through translations.

The call to `XtAddEventHandler` specifies which events trigger the event handler function. This is done with an *event mask* argument.* In an event mask, each bit represents one or more event types. Symbolic constants are defined by Xlib for each event mask bit. Multiple types of events can be selected at the same time by ORing together different masks with the bitwise OR operator (|). Note that there is not a one-to-one correspondence between event masks and event types. Some event masks select only one event, but others select multiple events. Furthermore, several of the masks select the same event type, but specify that it be delivered only when it occurs under special conditions. Table 8-1 shows the event masks and the event types they select.

*The event mask used in `XtAddEventHandler` is the same as the one used in the Xlib call `XSelectInput`.

Table 8-1. Event Masks and Event Types

Event Mask	Event Type
KeyPressMask	KeyPress
KeyReleaseMask	KeyRelease
ButtonPressMask	ButtonPress
ButtonReleaseMask	ButtonRelease
OwnerGrabButtonMask	n/a
KeymapStateMask	KeymapNotify
PointerMotionMask	MotionNotify
PointerMotionHintMask	
ButtonMotionMask	
Button1MotionMask	
Button2MotionMask	
Button3MotionMask	
Button4MotionMask	
Button5MotionMask	
EnterWindowMask	EnterNotify
LeaveWindowMask	LeaveNotify
FocusChangeMask	FocusIn
	FocusOut
ExposureMask	Expose
selected in GC by	GraphicsExpose
graphics_expose component)	NoExpose
ColormapChangeMask	ColormapNotify
PropertyChangeMask	PropertyNotify
VisibilityChangeMask	VisibilityNotify
ResizeRedirectMask	ResizeRequest
StructureNotifyMask	CirculateNotify
	ConfigureNotify
	DestroyNotify
	GravityNotify
	MapNotify
	ReparentNotify
	UnmapNotify
SubstructureNotifyMask	CirculateNotify
	ConfigureNotify
	CreateNotify
	DestroyNotify
	GravityNotify
	MapNotify
	ReparentNotify
	UnmapNotify
SubstructureRedirectMask	CirculateRequest
	ConfigureRequest
	MapRequest
(always selected)	MappingNotify

Table 8-1. Event Masks and Event Types (continued)

Event Mask	Event Type
(always selected)	`ClientMessage`
(always selected)	`SelectionClear`
(always selected)	`SelectionNotify`
(always selected)	`SelectionRequest`

The events that are always selected are called *nonmaskable events*. These can also be handled with an event handler, but not by specifying them in the event mask. An argument to the `XtAddEventHandler` call, *`non_maskable`*, is a Boolean value that, if `TRUE`, specifies that the event handler should be called for nonmaskable events. This event handler then must branch according to which of the seven types of nonmaskable events it is passed. A typical nonmaskable event handler is shown in Section 8.1.2.

8.1.1 Adding Event Handlers

Event handlers are added with a call to `XtAddEventHandler` or `XtInsertEventHandler`. `XtAddEventHandler` takes five arguments: the widget for which the handler is being added, an event mask, a flag that specifies whether or not this handler is for nonmaskable events (see below), the name of the handler, and optional client data. `XtInsertEventHandler` takes these and one additional argument: the position, either `XtListTail` or `XtListHead`.

A list of event handlers can be registered for the same event; but the same function will appear only once in the list with the same `client_data`. If the same function/`client_data` pair is registered again with `XtAddEventHandler`, nothing will happen except that the event mask for that function may change. But if the same function/`client_data` pair is registered again with `XtInsertEventHandler`, the function will be moved to the beginning or the end of the function list.

`XtAddEventHandler` or `XtInsertEventHandler` may be called before or after a widget is realized. In application code, this means the call can appear anywhere before `XtAppMainLoop`. In a widget, `XtAddEventHandler` or `XtInsertEventHandler` calls are placed in the `initialize` or `realize` methods.

Example 8-1 shows the code from *xterm* that registers an event handler for `FocusIn` and `FocusOut` events, and a gutted version of the event handler itself.

Example 8-1. Registering an event handler, and the handler function itself

```
extern void HandleFocusChange();

static void VTInitialize (request, new)
XtermWidget request, new;
{
    .
    .
    .
```

Example 8-1. Registering an event handler, and the handler function itself (continued)

```
    XtAddEventHandler(topLevel,      /* widget */
            FocusChangeMask,         /* event mask */
            FALSE,                   /* non-maskable events */
            HandleFocusChange,       /* event handler */
            (Opaque)NULL);           /* client_data */

    .
    .
    .
}

/*ARGSUSED*/
void HandleFocusChange(w, unused, event, continue_to_dispatch)
Widget w;
register XFocusChangeEvent *event;
XtPointer unused;                    /* client_data */
Boolean *continue_to_dispatch;
{
    if (event->type == FocusIn) {
                                     /* process FocusIn */
    .
    .
    .
    }
    else {
                                     /* process FocusOut */
    .
    .
    .
    }

    /*
     * If subsequent event handlers registered for this event
     * should not be called, set *continue_to_dispatch = False;
     * This is not recommended.
     */
}
```

In typical usage, either the event mask argument is a mask and the *non_maskable* argument is set to FALSE, or the event mask argument is set to zero and *non_maskable* is set to TRUE. Example 8-1 demonstrated the former case; now we'll look at the latter.

8.1.2 Adding Nonmaskable Event Handlers

The *non_maskable* argument of XtAddEventHandler specifies whether the specified event handler should be called in response to the events that can't be selected as described above. The nonmaskable events are GraphicsExpose, NoExpose, MappingNotify, SelectionClear, SelectionRequest, SelectionNotify, and ClientMessage. The first two of these events are selected using the graphics_exposure component of the GC, and the rest are always sent to the client

whenever they occur. `MappingNotify` is automatically handled by Xt, so it isn't passed to event handlers and you don't need to worry about it. The selection events are described in Chapter 10, *Interclient Communications*.

Because there are several nonmaskable event types, a nonmaskable event handler must be sure to branch according to the type of event, and throw away any event types not handled. You need not have all the code to handle all the types in a single event handler. Instead, you can handle each type in a separate handler, each registered separately. However, each handler would still need to check the event type because the entire list of them would be called for every nonmaskable event.

Example 8-2 shows the registration of a nonmaskable event handler and the handler itself, from *xman*.

Example 8-2. Adding a nonmaskable event handler

```
static void
Realize(w, valueMask, attributes)
register Widget w;
Mask *valueMask;
XSetWindowAttributes *attributes;
{
    .
    .
    .
    XtAddEventHandler(w, 0, TRUE,
            GExpose, NULL); /* Get Graphics Exposures */
} /* Realize */

/* ARGSUSED */
static void
GExpose(w, client_data, event)
Widget w;
XtPointer client_data;
XEvent *event;
{
    if (event->type == GraphicsExpose)
        Redisplay(w, event, NULL);    /*call the expose method directly*/
}
```

This event handler is sometimes used because Xt does not normally call the `expose` method in response to `GraphicsExpose` events. But in R4, another way to accomplish this has been introduced. If the `compress_exposure` field in the Core structure is set to `(XtExposeCompressMultiple | XtExposeGraphicsExpose)`, Xt will call the `expose` method with these events.

8.1.3 Removing Event Handlers

`XtRemoveEventHandler` takes the same arguments as `XtAddEventHandler`; if there are parameter mismatches, the call is quietly ignored. For example, the client data argument may be used to distinguish between different event handlers; if the client data argument does not match that which was passed in the `XtAddEventHandler`, then

XtRemoveEventHandler will do nothing. XtRemoveEventHandler is also silent about failing to remove a handler that was never added or a handler that was incorrectly specified.

8.1.4 Adding Raw Event Handlers

Xt also allows you to add event handlers without actually selecting events. The main purpose of this feature is for Xt to register functions before widgets are realized (because events can't be selected until windows are created during realization). Event handlers registered without selecting events are called *raw* event handlers, and are added with XtAddRawEventHandler or XtInsertRawEventHandler, which have the same calling sequences as XtAddEventHandler and XtInsertRawEventHandler. The event mask indicates which events the handler will be called in response to, but only when these events are selected elsewhere. Raw event handlers are supported mostly because they are used inside Xt. They are mentioned here only for completeness—you are unlikely to need them.

A raw event handler might be used to "shadow" another event handler (both added with the same event mask), such that until a primary event handler is added, the shadow handler will never be called. The primary handler will be added with XtAddEventHandler, which will select events, and then both handlers will be called when the appropriate events occur.

However, the "shadowing" technique is not necessary to assure that multiple calls to XtAddEventHandler don't result in wasted XSelectInput calls in which the event mask has not changed. Xt keeps a cache of the event masks of each widget, and calls XSelectInput only when it is necessary to change the window's event mask attribute in the server.

Raw event handlers are removed with a call to XtRemoveRawEventHandler.

8.2 Writing Routines That Use Specific Event Data

An event is a packet of information that the server sends to the client. Xlib takes this packet from the network and places it into an XEvent structure and places it on a queue until the client program requests it. Xt requests events from this queue and dispatches the event to the appropriate action routine or event handler for the widget in which the event occurred. The event itself is passed as an argument to the routine.*

Actually, XEvent is a C-language union of many event structures all the same size but with some different field names. The first member of the union, and of any of the individual event structures, is the event type. Table 8-2, later in this section, lists the event types and the matching event structure types.

*We'll look at the low-level routines Xt provides for directly manipulating the event queue later in this chapter.

Many action routines are intentionally written not to depend on the detailed information inside any particular type of event, so that the user can specify translations to call the action in response to different types of events. For example, it is useful for an action routine normally triggered by a pointer click to work when called in response to a key instead. Such an action should not depend on the event structure fields unique to button events.

However, many other action routines, and most event handlers, do use the detailed information inside event structures. The first member, `type`, identifies which type of event this structure represents, and hence implies which other fields are present in the structure.

To access event structure fields other than `type` you need to cast `XEvent` into the appropriate event structure type. If you are expecting only one type of event to trigger this action, then you can simply declare the argument as the appropriate type, as shown in Example 8-3.

Example 8-3. Casting the event structure by declaring action routine arguments

```
/*ARGSUSED*/
static void
ActionA(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    if ((event->type != ButtonPress) && (event->type != KeyPress)) {
        XtWarning("ActionA invoked by wrong event type.");
       /* possible exit here */
    }
        .
        .
}
```

When an action routine or event handler depends on the fields in a particular event structure, it is a good practice to check the event type in that action unless you are sure that the user can't change the translation (and thus the events used to invoke the action).

If you want the same code called for two event types, then you would do better to create two separate translations and two separate actions that each call a common routine. However, it is sometimes more convenient to have an action called by two different events. Example 8-4 shows the `ToggleCell` action from the BitmapEdit widget, which is called in response to either `MotionNotify` or `ButtonPress` events. This action inverts a pixel in the bitmap either if the pointer is clicked on a cell in the widget, or if it is dragged across the cell with the pointer buttons held down.

Example 8-4. Handling multiple event types in an action routine

```
static void
ToggleCell(w, event)
Widget w;
XEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    static int oldx = -1, oldy = -1;
    GC gc;
    int mode;
```

Example 8-4. Handling multiple event types in an action routine (continued)

```
    int newx, newy;

    if (event->type == ButtonPress) {
        newx = (w->bitmapEdit.cur_x + ((XButtonEvent *)event)->x) /
                w->bitmapEdit.cell_size_in_pixels;
        newy = (w->bitmapEdit.cur_y + ((XButtonEvent *)event)->y) /
                w->bitmapEdit.cell_size_in_pixels;
    }
    else if  (event->type == MotionNotify) {
        newx = (w->bitmapEdit.cur_x + ((XMotionEvent *)event)->x) /
                w->bitmapEdit.cell_size_in_pixels;
        newy = (w->bitmapEdit.cur_y + ((XMotionEvent *)event)->y) /
                w->bitmapEdit.cell_size_in_pixels;
    }
    else
        XtWarning("BitmapEdit: ToggleCell called with wrong event\
                type\n");
    .
    .
    .
}
```

Notice that some code is repeated to cast the event structure to the two different event types. With the current MIT implementation of Xlib, the positions of the `x` and `y` fields in the `XButtonEvent` and `XMotionEvent` structures are the same, and therefore this casting is unnecessary on many compilers. However, for strict ANSI C conformance these casts are necessary, and furthermore it is improper to depend on any particular implementation of Xlib. The order of the fields in one of these events could be different in some vendor's implementation of Xlib.

8.2.1 Event Types and Structure Names

Table 8-2 lists the event types and the matching event structure types. The event descriptions in the table will give you a general idea of what each event is for. Many of these events are not often used in applications, and more of them are automatically handled by Xt. We've already discussed how to use the most common event types and their abbreviations in translation tables in Chapter 7, *Events, Translations, and Accelerators*. Appendix C, *Event Reference* in Volume Five, *X Toolkit Intrinsics Reference Manual*, provides a complete reference to the circumstances under which each event is generated, what it is for, and the fields in each of the event structures. You will need this information to write action routines that use event-specific data.

Table 8-2. Event Types and Event Structures

Event Type	Structure	Description
KeyPress	XKeyPressedEvent	Key pressed
KeyRelease	XKeyReleasedEvent	Key released
ButtonPress	XButtonPressedEvent	Pointer button pressed
ButtonRelease	XButtonReleasedEvent	Pointer button released
KeymapNotify	XKeymapEvent	State of all keys when pointer entered
MotionNotify	XPointerMovedEvent	Pointer motion
EnterNotify	XEnterWindowEvent	Pointer entered window
LeaveNotify	XLeaveWindowEvent	Pointer left window
FocusIn	XFocusInEvent	This window is now keyboard focus
FocusOut	XFocusOutEvent	This window was keyboard focus
Expose	XExposeEvent	Part of window needs redrawing
GraphicsExpose	XGraphicsExposeEvent	Source of copy unavailable
NoExpose	XNoExposeEvent	Source of copy available
ColormapNotify	XColormapEvent	Window's colormap changed
PropertyNotify	XPropertyEvent	Property value changed
VisibilityNotify	XVisibilityEvent	Window has been obscured
ResizeRequest	XResizeRequestEvent	Redirect resize request to window manager
CirculateNotify	XCirculateEvent	Stacking order modified
ConfigureNotify	XConfigureEvent	Window resized or moved
DestroyNotify	XDestroyWindowEvent	Window destroyed
GravityNotify	XGravityEvent	Window moved due to win gravity attribute
MapNotify	XMapEvent	Window mapped
ReparentNotify	XReparentEvent	Window reparented
UnmapNotify	XUnmapEvent	Window unmapped
CirculateRequest	XCirculateRequestEvent	Redirect stacking order change to window manager
ConfigureRequest	XConfigureRequestEvent	Redirect move or resize request to window manager
MapRequest	XMapRequestEvent	Redirect window map request to window manager
MappingNotify	XMappingEvent	Keyboard mapping changed
ClientMessage	XClientMessageEvent	Client-dependent
SelectionClear	XSetSelectClearEvent	Current owner is losing selection
SelectionNotify	XSelectionEvent	Selection is ready for requestor
SelectionRequest	XSelectionRequestEvent	Request for selection to current owner

8.3 File, Pipe, and Socket Input

`XtAppAddInput` allows a program to obtain input from a file. This is not merely reading the file once, but monitoring it for further activity. Under UNIX this can be used to get input from pipes and sockets, since they are variations of files. We will demonstrate getting file and pipe input in this section.

The `XtAppAddInput` routine takes four arguments: a file descriptor, a flag (see below), your function, and `client_data`.

`XtAppAddInput` returns an ID that uniquely identifies the `XtAppAddInput` request. You can use the ID to cancel the request later with `XtRemoveInput`.

One argument of `XtAppAddInput` is a file descriptor (this file must be open before calling `XtAppAddInput`). Since implementation of files varies between operating systems, the actual contents of the parameter passed as the file descriptor argument to these routines is operating system-dependent. Therefore, this code is inherently nonportable.

Possible values for the mask and their meanings are as shown in Table 8-3.

Table 8-3. Other Input Source Masks

Mask	Description
`XtInputReadMask`	File descriptor has data available
`XtInputWriteMask`	File descriptor available for writing
`XtInputExceptMask`	I/O errors have occurred (exceptions)
`XtInputNoneMask`	Never call function registered

Calling these argument values masks is something of a misnomer, since they *cannot* be ORed together. However, you can call `XtAppAddInput` additional times to register a separate function (or the same function) for each of these masks on the same file descriptor.

8.3.1 Getting File Input

In Example 8-5, a program called *xfileinput* reads new characters from a file whenever they appear. In other words, the program will initially print to the standard output the contents of the file specified on the command line, and it will print any characters that are later appended to that file. Try the program *xfileinput* as follows:

```
echo "test string" > testfile
xfileinput testfile &
echo "more text" >> testfile
```

A program such as this functions similarly to the UNIX command *tail -f*. It could be used to monitor system log files, or other similar files that grow.

The code shown in Example 8-5 opens the file and calls `XtAppAddInput` in `main`. The `get_file_input` function registered with `XtAppAddInput` reads and prints characters from the file.*

Example 8-5. Getting file input with XtAppAddInput

```
/* header files */
  .
  .
  .

/* ARGSUSED */
get_file_input(client_data, fid, id)
XtPointer client_data;    /* unused */
int *fid;
XtInputId *id;
{
    char buf[BUFSIZ];
    int nbytes;
    int i;

    if ((nbytes = read(*fid, buf, BUFSIZ)) == -1)
        perror("get_file_input");

    if (nbytes)
        for (i = 0; i < nbytes; i++)
            putchar(buf[i]);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, goodbye;
    FILE *fid;
    String filename;

    topLevel = XtVaAppInitialize(&app_context, "XFileInput", NULL,
            0, &argc, argv, NULL, NULL);

    if (argv[1] == NULL) {
        fprintf(stderr, "xfileinput: filename must be specified on\
                command line.\n");
        exit(1);
    }

    filename = argv[1];
      .
      .
      .

    /* open file */
    if ((fid = fopen(filename, "r")) == NULL)
        fprintf(stderr, "xfileinput: couldn't open input file.\n");

    /* register function to handle that input, NULL arg
     * is client_data */
```

*Note that the code for opening and reading files is probably not portable to operating systems other than UNIX.

Example 8-5. Getting file input with XtAppAddInput (continued)

```
    XtAppAddInput(app_context, fileno(fid), XtInputReadMask,
            get_file_input, NULL);

    XtRealizeWidget(topLevel);

    XtAppMainLoop(app_context);
}
```

The function registered with `XtAppAddInput` is called with *`client_data`* (used for passing in any application data), a pointer to the file descriptor, and the ID of the `XtAppAddInput` request. You can use a call to `XtRemoveInput` in the function registered with `XtAppAddInput` if that function is only to be called once. One argument of the `XtRemoveInput` call is the ID of the `XtAppAddInput` request.

Under some operating systems, the function registered with `XtAppAddInput` is called very frequently even when no new input is available. This is because Xt makes a system call to detect whether the file is "ready for reading," and some operating systems say the file is ready even when there is nothing to read. The example shown above works, but it loads down the system much more than necessary. You may wish to check the file every quarter second instead of continuously, by adding and removing your input handler periodically using timeouts (as described in Section 8.4). The same goes for input from a pipe.

8.3.2 Getting Pipe Input

The code to get pipe input is almost identical to the code just shown that gets file input. The only difference is that we use *popen* instead of *fopen*, and change the various error messages. Now instead of treating the command-line argument as a filename, it is treated as a program run under a shell. This program's output is piped into our application. For example, here is an example of how to invoke this version of *xpipeinput*:

```
spike% xpipeinput "cal 11 1989"
    November 1989
 S  M Tu  W Th  F  S
             1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

(Program continues to monitor pipe for further input until application exits.)

Note that *xpipeinput* is reading the string "cal 11 1989" from the command line, invoking a shell, running the command specified by the string under this shell, reading the output of the shell, and then printing it on the standard output. This is an easy way to use all kinds of shell scripts and utilities from within a program.

If you want your application to accept standard input, this is even easier. Remove the code that reads the filename from the command line and remove the `popen` call to open the pipe, since the pipe from `stdin` is always open. Then use the `XtAppAddInput` function shown in Example 8-6.

Example 8-6. Reading stdin from an Xt application

```
XtAppAddInput(app_context, fileno(stdin), XtInputReadMask,
        get_file_input, NULL);
```

Once you have done this, you can invoke *xpipeinput* as follows:

```
spike% cal 11 1989 | xpipeinput
    November 1989
 S  M Tu  W Th  F  S
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

(*Program continues to monitor pipe for further input until application exits.*)

Note that in this case, *xpipeinput* is reading directly from stdin, and then printing the output to stdout. With more code, it could display this calendar in a Text widget instead.

Also note that *xpipeinput* loads down the system for the same reasons as *xfileinput*, described at the end of the last section. Solutions to this problem are also described there.

8.4 Timeouts

A program may wish to be notified when a period of time has elapsed, while being able to do other things in the meantime. For example, a clock widget requires a periodic nudge to change the time it is displaying, but must also be able to redisplay itself at any time in case of exposure.

This is done by using `XtAppAddTimeOut`. This routine is passed a time interval in milliseconds, and the address of a function to be invoked when the time interval expires. As usual, a `client_data` argument can also be registered. The `XtAppAddTimeout` routine returns a handle that can be used to cancel the timeout before it triggers, if necessary.

A timeout is automatically removed when the registered function is called. Therefore, to have a function called repeatedly, every *N* milliseconds, the registered function must add the timeout again by calling `XtAppAddTimeOut`.

One of the major applications of timeouts other than clocks is in real-time games. Figure 8-1 shows the appearance of a game called *xtetris* after it has been played for a couple of minutes.

The object of the game is to steer falling blocks and rotate them so that they fit well into the existing fallen blocks.* The game is over when the blocks pile up to the top of the window.

*This game is provided with the example source code. It is an X version of a game available on the Macintosh called *Tetris*, trademark of AcademySoft-ELORG, copyright and trademark licensed to Andromeda Software Ltd. The original concept of the game is by Alexi Pazhitnov and Vadim Gerasimov.

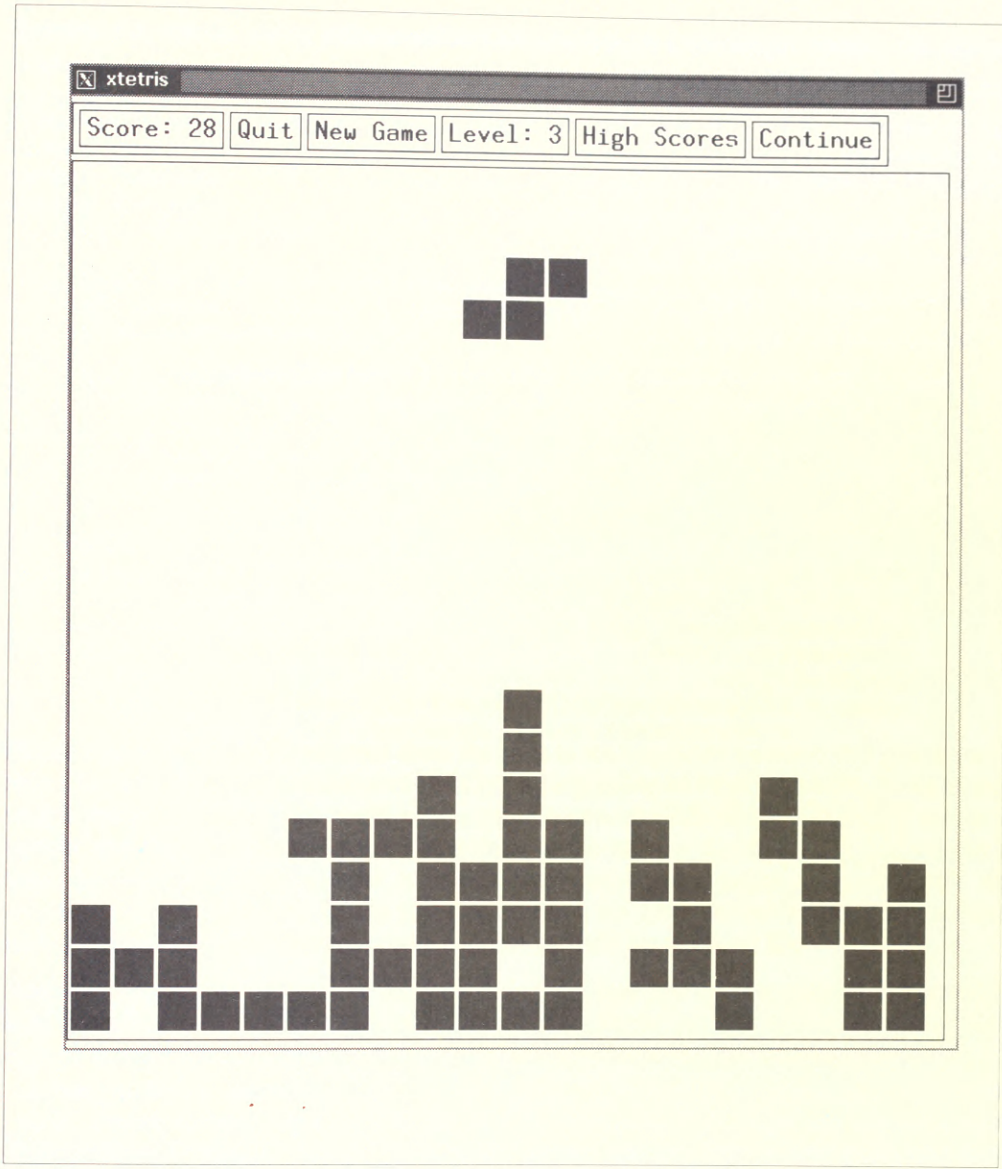


Figure 8-1. xtetris in play

Every time a row is completely filled, it is removed and all the blocks above it move down one row. The window in which the blocks fall is a specialized widget. This game uses timeouts to time the falling of the blocks.

Example 8-7 is an excerpt from a widget used by *xtetris* that adds the timeout. The timeout function itself is also shown.

Example 8-7. xtetris: registering a timeout and the timeout function

```
static XtIntervalId timer;

static void
StartBlock(w)
BitmapEditWidget w;
{
    w->bitmapEdit.cur_x = 9;
    w->bitmapEdit.cur_y = 0;

    w->bitmapEdit.type = PickType();

    DrawBlock(w, DRAW);

    timer = XtAppAddTimeOut(XtWidgetToApplicationContext(w),
            w->bitmapEdit.delay, MoveDown, w);
}

static void
MoveDown(w)
BitmapEditWidget w;       /* client_data */
{
    if (CanMoveDown(w)) {
        drawBlock(w, UNDRAW);
        w->bitmapEdit.cur_y++;
        DrawBlock(w, DRAW);
        CopyBlock(w);
        timer = XtAppAddTimeOut(XtWidgetToApplicationContext(w),
                w->bitmapEdit.delay, MoveDown, w);
    }
    else { /*block has hit bottom or other stationary blocks*/
        UpdateCellArray(w);
        KillRows(w);
        Score(w);
        w->bitmapEdit.delay -= 5;
        StartBlock(w);
    }
}
```

Notice that in widget code, the application context is specified using `XtWidgetToApplicationContext`.

Notice also that a timeout function is called with only one argument, *`client_data`*. Inside a widget, this argument is commonly used to pass in the widget instance pointer. Also notice that every time a block hits the bottom, the instance variable `delay` is decremented by 5, which reduces the number of milliseconds of delay used when `XtAppAddTimeOut` is next called. In other words, the blocks fall progressively faster.

xtetris also needs to remove a timeout in one of its routines. The user can "drop" a block to score extra points (if there is enough time). Whenever a block is dropped, the block is immediately moved down as far as it will go, and a new block is started. If the `Drop` action did not remove the timeout, the new block would be started with a new timeout while an existing timeout was already in force. This would mean that the `MoveDown` timeout function would be invoked twice in quick succession when each of these timeouts expired. Example 8-8 shows the `XtRemoveTimeOut` call in the `Drop` action.

Example 8-8. xtetris: calling XtRemoveTimeOut

```
/*ARGSUSED*/
static void
Drop(tw, tevent, params, num_params)
Widget tw;
XEvent *tevent;
String *params;
Cardinal *num_params;
{
     TetrisWidget w = (TetrisWidget) tw;
     XButtonEvent *event = (XButtonEvent *) tevent;
     XtRemoveTimeOut(timer);

     while (CanMoveDown(w)) {
         DrawBlock(w, UNDRAW);
         w->bitmapEdit.cur_y++;
         DrawBlock(w, DRAW);
         CopyBlock(w);
     }

     UpdateCellArray(w);
     KillRows(w);
     score++;
     Score(w);
     w->bitmapEdit.delay -= 5;
     StartBlock(w);
}
```

Notice that the `timer` ID returned from the calls to `XtAppAddTimeOut` is a global variable. Xt calls the timeout function with only one argument, and that argument passes in the widget instance pointer. We could have created a structure containing the widget instance pointer and the timer ID and passed its pointer to the timeout function. But this wouldn't help, because the action routine in which we remove the timeout is passed with no *`client_data`* argument. (It has string parameters, but these are hardcoded in the actions table.) Therefore, we are forced to have a global variable for the timer ID.

Note that between the time when the timeout is registered and when it triggers, the application processes events in `XtAppMainLoop`. Therefore, all the widget's actions and `expose` method are in operation between the invocations of the timeout function.

8.4.1 Visibility Interest

Timeouts operate regardless of the visibility of the application. Since it is pointless for most games to continue operating while obscured, it makes sense to remove the game's timeouts when the game is partially or fully obscured (or iconified). To do this, you can set the `visible_interest` field in the Core class structure to `TRUE`, and then check the `visible` field of the Core instance structure periodically. When the application is fully obscured, you add a separate timeout to continue testing the visibility status. When the visibility status is satisfactory once again, the game can add its timeout again. All these changes are in the widget's *.c* file. First we set the `visible_interest` field to `TRUE` in the Core structure:

```
BitmapEditClassRec bitmapEditClassRec = {
    /* core_class fields */
    .
    .
    .
    /* visible_interest              */ TRUE,
    .
    .
    .
}
```

Second we change:

```
timer = XtAppAddTimeOut(XtWidgetToApplicationContext
    (w), w->bitmapEdit.delay, MoveDown, w);
```

to:

```
if (w->core.visible == FALSE)
            timer = XtAppAddTimeOut(XtWidgetToApplicationContext
                (w) 250, CheckVisibility, w);
        else
            timer = XtAppAddTimeOut(XtWidgetToApplicationContext
(w)
w->bitmapEdit.delay, MoveDown, w);
```

And finally, we add the timeout function that continues to check the visibility status.

```
static void
CheckVisibility(w)
BitmapEditWidget w;     /* client_data */
{
    if (w->core.visible == FALSE)
        timer = XtAppAddTimeOut(250, CheckVisibility, w);
    else
        timer = XtAppAddTimeOut(w->bitmapEdit.delay,
                MoveDown, w);
}
```

Unfortunately, the Core `visible` field is `TRUE` even if a tiny sliver of the widget is visible. The only way to get around this is to add an event handler (or translation) for `VisibilityNotify` events and to add an instance variable to maintain the visibility state. The event handler or action would check the `state` field of the event, and put the game into hibernation if the window is only partially obscured. However, this approach has the opposite problem; it disables the game even when only a sliver is obscured.

There is nothing you can do about the game continuing to run while being moved or resized with the window manager. However, using the Core `visible_interest` field does stop the game when it is iconified.

8.5 Work Procedures

A work procedure is an application-supplied function that is executed while an application is idle waiting for an event. Work procedures are registered with `XtAppAddWorkProc`. They can perform any calculation that is short enough that the routine will return in a small fraction of a second. If the work procedure is too long, the user's response time will suffer.

If a work procedure returns `TRUE`, then Xt will remove it and it will not be called again. But if one returns `FALSE`, it will be called repeatedly every time there is idle time, until the application calls `XtRemoveWorkProc`. A work procedure would return `TRUE` if it performs a one-time setup such as creating a pop-up widget. It would return `FALSE` if it were continuously updating a disk file as security against a system crash or server connection failure.

You can register multiple work procedures, and they will be performed one at a time. The most recent work procedure added has the highest priority. Therefore, for example, if you want to create ten pop-up widgets during idle time, you should add ten work procedures. The pop up that you expect to need first should be added in the last work procedure registered.

The call to register a work procedure is shown in Example 8-9.

Example 8-9. Registering an Xt work procedure

```
static Boolean create_popup();
   .
   .
   .
main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    XtWorkProcId popup_work_ID;
    Widget topLevel;
        .
        .
    /* XtAppInitialize, create widgets, etc. */
        .
        .
    popup_work_ID = XtAppAddWorkProc(app_context,
            create_popup, topLevel);
        .
        .
    XtRealizeWidget(topLevel);
    XtAppMainLoop(app_context);
}
```

Notice that `XtAppAddWorkProc` returns an ID of type `XtWorkProcId`, which is used only in any subsequent call to `XtRemoveWorkProc`. You can cast the returned value to `void` if you do not intend to explicitly remove the work procedure.

The *`client_data`* argument passes application data into the work procedure. It is used just like the same argument in callback functions. Example 8-10 shows a work procedure to create a pop-up widget.

Example 8-10. A work procedure to create a pop-up widget

```
Widget pshell;

/* work procedure */
Boolean
create_popup(client_data)
XtPointer client_data;
{
    Widget parent = (Widget) client_data;
    Widget dialog, dialogDone;

    pshell = XtCreatePopupShell(
            "pshell",
            transientShellWidgetClass,
            parent,
            NULL,
            0
            );

    dialog = XtCreateManagedWidget(
            "dialog",              /* widget name */
            dialogWidgetClass,      /* widget class */
            pshell,              /* parent widget */
            NULL,                  /* argument list */
            0                      /* arglist size */
            );

    dialogDone = XtCreateManagedWidget(
            "dialogDone",          /* widget name */
            commandWidgetClass,     /* widget class */
            dialog,              /* parent widget */
            NULL,                  /* argument list */
            0                      /* arglist size */
            );

    XtAddCallback(dialogDone, XtNcallback, DialogDone, dialog);

    return(True);     /* makes Xt remove this work proc automatically */
}
```

Remember that Xt cannot interrupt a work procedure while it is running; the procedure must voluntarily give up control by returning, and it must do so quickly to avoid slowing user response.

If your application has any big jobs that it must do, the only way to do them without resulting in long delays is to write the code that does the big job in a way that voluntarily interrupts itself and saves its state so that it can be restarted where it left off. One way to run such a task is as a work procedure, but this is only useful for tasks that need not be done before other application tasks can begin. If you want `Expose` processing to continue but no other application task to begin until your task is done, you would use the same type of code but place low-level event management routines in it, or make the rest of the application insensitive until the task is done.

8.6 Low-level Management of the Event Queue

As you know, an X Toolkit application simply calls `XtAppMainLoop` to begin processing events. `XtAppMainLoop` itself is quite simple: it consists of an infinite loop calling two lower-level routines, `XtAppNextEvent` and `XtDispatchEvent`. `XtAppNextEvent` extracts the next event from the application's event queue; `XtDispatchEvent` actually uses the event to invoke the appropriate actions or event handlers. (The functions registered by `XtAppAddInput` and `XtAppAddTimeOut` are dispatched directly by `XtAppNextEvent`; if no events are available, `XtAppNextEvent` flushes the X output buffer, and calls any work procedures registered by `XtAppAddWorkProc`.)

An application can provide its own version of this loop, as shown in Example 8-11. For example, it might test some application-dependent global flag or other termination condition before looping back and calling `XtAppNextEvent`. Or for fine-grained debugging, it might be worthwhile to insert a routine that prints out the type of each event dispatched.

Example 8-11. Skeleton of a custom main loop

```
void MyMainLoop(app_con)
XtAppContext app_con;
{
     XEvent event;

     for (;;) {
         XtAppNextEvent(app_con, &event);
         XtDispatchEvent(&event);

    /* Do application-specific processing here */

     }
}
```

8.6.1 XtPending and XtPeekEvent

All event sources depend on idle time in the application to return to `XtAppMainLoop` where Xt can check to see if input is available from any of the various sources. If an application has long calculations to make, the program may not return to `XtAppMainLoop` frequently enough to detect important input in a timely fashion. The application itself should, if possible, suspend lengthy calculations for a moment to check whether input is available. Then it can determine whether to process the input before continuing, or finish the calculation.

To detect whether input from any input source is available, you can call `XtPending`. This function returns a mask composed of a bitwise OR of the symbolic constants `XtIMXEvent`, `XtIMTimer`, and `XtIMAlternateInput`. These constants refer to X events, timer events, and alternate input events, respectively.

To find out what the first event in the queue contains, you can call `XtPeekEvent`. This function returns an event structure without removing the event from Xlib's queue.

It is also possible to remove and process a single event. XtAppProcessEvent combines some (but not all) of the functions from XtAppNextEvent and XtDispatchEvent. That is, while XtAppNextEvent takes the next event from the queue, whatever it is, XtAppProcessEvent allows you to specify as a mask a bitwise OR of the symbolic constants XtIMXEvent, XtIMTimer, and XtIMAlternateInput. This lets you select only some of these event types for processing. In addition, XtAppProcessEvent actually calls XtDispatchEvent to dispatch X events, so only this one call is necessary.

8.6.2 Event Filters

As you saw in Chapter 5, *Inside a Widget*, the class structure contains three Boolean fields that control Xt's event filters. These are compress_motion, compress_enterleave, and compress_exposure. Widgets set these fields to TRUE when repeated events of these types are unwanted. Each would be used in different situations. If turned on, they tell Xt to search Xlib's queue for a certain event sequence and then remove repeated occurrences of those events from the queue.

When the compress_motion filter is set to TRUE, and there is a series of MotionNotify events on the queue (which occurs when the application gets behind in processing them), the filter throws out all but the last one (the most recent position). This is useful for widgets that need the most up-to-date position but do not need a complete history of pointer positions.

The compress_enterleave filter throws out all EnterNotify/LeaveNotify pairs on the same window in which there are no intervening events. This would be used by a widget that is interested in enter and leave events, but not if the application falls behind. For example, even the Command widget sets compress_enterleave to TRUE. It highlights its border when the pointer enters, and clears it when the pointer leaves. But if for some reason the widget falls behind and has not highlighted the border by the time the LeaveNotify event arrives with no intervening events, the border will not be highlighted. To see this, move the pointer quickly across a large panel of Command widgets such as in *xmh*, and you will see that not all of them draw and then undraw the border.

The symbols used for setting the compress_exposure filter have changed in R4. If the field of this name in the Core class structure is set to FALSE or XtExposeNoCompress, a widget's expose method is called once in response to each Expose event in a contiguous series. Each event specifies a different rectangle of the widget that needs redrawing.

With compress_exposure set to TRUE or XtExposeCompressSeries, however, a contiguous series of events resulting from one user action is compressed into a single modified Expose event and the expose method is called only once. This modified Expose event contains the bounding rectangle of the union of all the rectangles in the individual events. In this case the expose method is also passed an Xlib Region that describes in detail the area exposed. Probably the most useful value for compress_exposure is XtExposeCompressMultiple, which compresses all the contiguous events resulting from multiple contiguous user actions.

When ORed with any of the XtCompress* symbols, the XtExposeGraphicsExpose symbol causes Xt to call the expose method with any GraphicsExpose events that occur. Remember that you must set the graphics_exposures component to True in the GC used in XCopyArea or XCopyPlane in order to get GraphicsExpose events. XtExposeGraphicsExposeMerged, when ORed with an XtCompress* symbol, merges contiguous Expose and GraphicsExpose events together before calling the expose method.

The XtExposeNoExpose symbol causes Xt to dispatch NoExpose events to the expose method. This doesn't make much sense; if you need NoExpose events it is better to add an event handler or translation to handle them.

The remaining symbol is XtExposeCompressMaximal. This symbol is dangerous and should not be used: it merges non-continuous Expose events into one event before calling the expose method. This is unwise because the intervening events could be ConfigureNotify events that change the size of the window. When this happens, the application will redraw itself, then receive the ConfigureNotify, but then it will not redraw itself in the new size because the Expose event that would trigger the drawing has already been removed from the queue.

Almost all widgets except those that display a large amount of text should set this filter to XtExposeCompressMultiple. Text widgets can very efficiently redraw only the needed parts of the window because each character is in a fixed location. (Characters are in fixed locations in the Text widget because it uses fixed-width fonts—this is not applicable to widgets that display proportional fonts.) Therefore, it can efficiently process all the Expose events one at a time.

8.6.3 Input Sensitivity

There are times when some widgets should be insensitive to events in which they are usually interested. For example, a Command widget should be insensitive when the command that it executes is already in operation.

Widget sensitivity is inherited. For example, if a parent widget is insensitive, then its children are too. In other words, an entire box full of widgets can be set insensitive by simply setting the box widget insensitive. Note, however, that this process can be a little slow because all the widgets in the box that honor sensitivity will redraw themselves dimmed or grayed. A widget is made insensitive from an application by calling XtSetSensitive with the sensitive argument set to FALSE, or using XtVaSetValues on the XtNsensitive resource (XtSetSensitive is slightly faster).

Any widget that may need to be disabled for a time by the application should change its visible appearance when insensitive.

The widget that has one of the XtCallback* standard pop-up callback functions registered on its callback list will automatically be set insensitive when the callback is triggered. If the XtCallbackPopdown callback function is registered on this widget it will automatically be set sensitive again when this callback is invoked.

9

Resource Management and Type Conversion

This chapter is a more thorough discussion of how resources work and how they should be used. This chapter describes in detail the resource file format and the rules that govern the precedence of resource settings. It also describes how to add your own type converter so that you can have application or widget-specific data set through resources. Finally, it describes subresources and how to use them.

In This Chapter:

Review of Resource Fundamentals 248
How the Resource Database Works 251
Form of Resource Specifications 252
Merging of Resource Files 253
The Language String 255
Fallback Resources 257
Resource Matching Algorithm 258
Resource Precedence Rules 260
Type Conversion 263
Conversions from XtRString 263
Other Built-in Type Conversions 266
Special Resource Defaults That Do Not Use Conversion 267
Registering Type Converters 269
Passing Arguments to a Type Converter 272
Explicitly Invoking a Converter 274
Writing a Type Converter 275
Defining the Default Value 277
Subparts and Subresources 278
The Hook Methods 279
Managing Subresources 279

9
Resource Management and Type Conversion

This chapter provides a thorough discussion of how resources work and how they should be used. First, we describe how to define resources, the complete syntax of resource files, and the rules that describe the precedence of one resource setting over another. For the sake of completeness, and to make sure that the ideas are presented in context, there is some repetition of material that has been presented earlier.

Next, the chapter describes the resource conversions performed automatically by Xt. As you may recall from the discussion in Chapter 2, *Introduction to the X Toolkit*, a value converter is invoked by Xt to convert a resource from the string form specified in resource files to the representation type actually used in the application or widget. For the representation types understood by Xt, simply listing the representation symbol (a constant beginning with `XtR`) in the resource list is enough to make Xt automatically perform the conversion. But if you create a representation type unknown to Xt, you need to write a type converter routine and register it with Xt before the automatic conversion can take place. We discuss both the standard converters and how to write a new one.

Finally, the chapter describes a mechanism Xt provides whereby widgets or applications may have subparts with separate sets of resources. Special routines are provided for setting and getting these resources. The R3 Athena Text widget used subparts to implement replaceable units that provide the data storage and display for text data. This allowed the same central code to edit a disk file or a string. But using subparts is now out of favor; the R4 Text widget uses objects to accomplish the same modularity (see Chapter 13, *Miscellaneous Toolkit Programming Techniques*).

9.1 Review of Resource Fundamentals

As we've previously discussed, widgets and applications can declare some or all of their variables as resources. Not every variable need be a resource—only those for which values need to be supplied by the user (or for a widget, also by the application programmer) through the Resource Manager. Both applications and widgets may use nonresource variables for internal bookkeeping, or for storing values calculated or otherwise derived from resources.

Resources are defined using an `XtResource` structure, which is declared as follows:

```
typedef struct {
    String resource_name;      /* specify using XtN symbol */
    String resource_class;     /* specify using XtC symbol */
    String resource_type;      /* actual data type of variable */
    Cardinal resource_size;    /* specify using sizeof() */
    Cardinal resource_offset; /* specify using XtOffsetOf() */
    String default_type;       /* will be convrted to resrce_type */
    XtPointer default_address;/* address of default value */
} XtResource, *XtResourceList;
```

For example, Example 9-1 shows two of the resources defined by the Athena Label widget:

Example 9-1. Two resources defined by the Athena Label widget

```
static XtResource resources[] = {
    {
    XtNforeground,  /* Resource name is foreground */
    XtCForeground,  /* Resource class is Foreground */
    XtRPixel,       /* Resource type is Pixel */
    sizeof(Pixel),  /* allocate enough space to hold a Pixel value */
    XtOffsetOf(LabelRec, label.foreground),/*where in instnce strct*/
    XtRString,      /*Default val is a String (will need conversion)*/
    XtDefaultForeground  /* Address of default value */
    },
                    .
                    .
                    .
    {
    XtNlabel,
    XtCLabel,
    XtRString,
    sizeof(String),
    XtOffsetOf(LabelRec, label.label),
    XtRString,
    NULL
    },
                    .
                    .
                    .
}
```

The fields in the `XtResource` structure are used as follows:

- The resource name is usually similar to the name of the variable being set by the resource; by convention, it begins with a lower-case letter, and no underscores are used to separate multiple words. Instead, the initial character of subsequent words is given in

upper case. For example, the resource name for a variable named `border_width` would be `borderWidth`, and the defined constant used to refer to this name would be `XtNborderWidth`.

As described previously, the name, class, and representation type of resources are specified in the resource list (and elsewhere in Xt code, but not in user database files) using symbolic constants defined in *<X11/StringDefs.h>*, and consist of the actual name, class, or type preceded by the characters `XtN`, `XtC`, or `XtR`, respectively. Use of these constants provides compile-time checking of resource names, classes, and types. Without the constants, a misspelling would not be noticed by the compiler, since resource names, classes, and representation types are simply strings. The misspelling would be considered a real resource at run time. Nothing would happen if it were set from the application, because no widget would actually use it. If, on the other hand, the misspelling were in the widget resource list, the application's setting of the intended resource would have no effect.

Newly-defined resources may use a name, class, or type constant defined in *<X11/StringDefs.h>*, if an appropriate one exists. Otherwise, the constant is defined in the widget's public header file, or for application resources, in the application itself, or in the application header file, if any.)

- For many resources, the class name is simply the same as the resource name, except that the `XtC` prefix is used, and, by convention, the first letter of the name is capitalized. For example, the class name constant for the `XtNbackgroundPixel` resource is `XtCBackgroundPixel`. However, when appropriate, a single class can be used for a group of related resources. This allows a single setting in the resource database to control the value of multiple resources.

  For example, a widget can have several elements that use pixel values (i.e., colors) as resource settings: background, foreground, border, block cursor, pointer cursor, and so on. Typically, the background defaults to white and everything else to black. If the `background` resource has a class of `Background`, and all the other pixel resources a class of `Foreground`, then a resource file needs only two lines to change all background pixels to offwhite and all foreground pixels to darkblue:

```
*Background:      offwhite
*Foreground:      darkblue
```

- The representation type of the resource is specified by the `resource_type` field of the resource list, using a symbolic constant prefixed by `XtR`. Table 9-1 lists the correspondence between the `XtR` symbols defined by Xt, and actual C data types or X data types and structures.

Table 9-1. Resource Type Strings

Resource Type	Data type
XtRAcceleratorTable	XtAccelerators
XtRAtom	Atom
XtRBitmap	Pixmap (of depth one)
XtRBoolean	Boolean
XtRBool	Bool
XtRCallback	XtCallbackList
XtRCallProc	*see final bullet below*
XtRCardinal	Cardinal
XtRColor	XColor
XtRColormap	Colormap
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Display *
XtREnum	XtEnum
XtRFile	FILE *
XtRFloat	float
XtRFont	Font
XtRFontStruct	XFontStruct *
XtRFunction	(*)()
XtRGeometry	String - format as defined by XParseGeometry
XtRImmediate	*see final bullet below*
XtRInitialState	int
XtRInt	int
XtRLongBoolean	long
XtRObject	Object
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	XtPointer
XtRPosition	Position
XtRScreen	Screen *
XtRShort	short
XtRString	char *
XtRStringArray	String *
XtRStringTable	char **
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRVisual	Visual *
XtRWidget	Widget
XtRWidgetClass	WidgetClass
XtRWidgetList	WidgetList
XtRWindow	Window

As we'll discuss in detail in Section 9.3, Xt automatically converts values in the resource database (which always have the type `XtRString`, since resource files are made up entirely of strings) into the target type defined by `resource_type`.

- The `resource_size` field is the size of the resource's actual representation in bytes; it should always be specified as `sizeof(`*`type`*`)` (where *`type`* is the C-language type of the resource) so that the compiler fills in the value.
- The `resource_offset` field is the offset in bytes of the field within the widget instance structure or application data structure. The `XtOffsetOf` macro is normally used to obtain this value. This macro takes as arguments the data structure type, and the name of the structure field to be set by the resource.
- If no value is found in the resource database, the value pointed to by the `default_address` field will be used instead. The type of this default value is given by the `default_type` field. If the `default_type` is different from the `resource_type`, a conversion will be performed automatically in this case as well.

  There are two special resource types that can be used only as the `default_type`. `XtRImmediate` means that the value in the `default_address` field is to be used as the actual resource value, rather than as a pointer to it (or in the case of a string, the value is a pointer to a string, instead of a pointer to a pointer to a string). The other special resource type, `XtRCallProc`, is a pointer to a function that will supply the default value at run time. We'll demonstrate the use of these values in Section 9.3.3.

9.2 How the Resource Database Works

Xt's resource handling is based on the resource manager built into Xlib, but Xt adds a great deal. While using the resource manager from Xlib is cumbersome, from Xt it is easy: to use resources in existing widgets, all you have to do is write the application-defaults file.

Xt's handling of resources occurs in two stages:

1. When the application starts up, with a call to `XtAppInitialize`, Xt reads the application-defaults file, along with several other resource files, command line options, and the RESOURCE_MANAGER property stored in the server by the user with *xrdb*. (Any, all, or none of these may contain data.) It merges all these sources of data into one internal database that is used when each widget is created.
2. Whenever you create a widget, the call to `XtVaCreateManagedWidget` reads the resource database and automatically sets widget resources to the values in the database. In order to explain this stage more clearly, we further divide it into two separate steps in the sections that follow. First, Xt compares the settings in the database to the widget's class and instance hierarchy, to find which settings apply to the widget being created. Second, Xt decides which of the (possibly conflicting) settings that apply to that widget should actually be used.

If the value of a resource is hardcoded by passing arguments to `XtVaCreateManagedWidget` or `XtVaSetValues`, the hardcoded value overrides the value looked up from the resource database.

To retrieve the value of application resources from the database, an application must make an explicit call to `XtGetApplicationResources`, as described in Section 3.5.3.

9.2.1 Form of Resource Specifications

As discussed in Chapter 2, each entry in the merged database (and in the source databases) is a resource specification/value pair. For application resources, the specification is the application name followed by a period and the resource name. The value to which the resource is to be set follows, after a colon and optional white space.* For example:

```
xterm.scrollBar:  on
```

An asterisk can be used as a "wildcard" in place of the application name. For example:

```
*scrollBar:  on
```

would set a resource named `scrollBar` to "on" in any application that recognized a resource of that name.

For widget resources, the specification leading up to the resource name may contain a widget instance or class hierarchy (or a mixed instance/class hierarchy). Some examples are shown below. (Remember that instance names begin with a lower case letter, while class names begin with an upper case letter.)

specification	*value*	
`xbitmap.box.quit.label:`	`Quit`	*fully-specified instance hierarchy*
`XBitmap.Box.Command.Foreground:`	`blue`	*fully-specified class hierarchy*
`XBitmap.Box.quit.foreground:`	`blue`	*mixed class and instance hierarchy*

An instance hierarchy describes the instance names of the widget's ancestors. A class hierarchy describes the class names of the widget's ancestors. This portion of the resource specification may consist of a mixture of instance names and class names (each of which describes one generation in the widget's hierarchy), separated by periods or asterisks.

- A period (.) is referred to as a *tight* binding.
- An asterisk (*) is referred to as a *loose* binding.

A tight binding means the left component must be the parent of the right component in the instance hierarchy. A loose binding means the left component must only be an ancestor of the right component; there can be any number of levels in the hierarchy between the two.

*Note the distinction between what we are calling the resource *specification* (the fully qualified name of the resource, up to the colon), and the *value* (the actual value to which the resource is to be set). We refer to both the specification and the value together as a *resource setting*.

Loose bindings are preferable because they stand a better chance of working when the instance hierarchy changes. Tight bindings are rarely necessary at every position in the resource specification, since widget names are usually unique and single widgets can be identified by name. Furthermore, it takes more text to specify the complete instance hierarchy for every widget to be set.

Using loose bindings, the instance, class, or instance/class hierarchy may be abbreviated to the point where specifying the hierarchy as a single asterisk would indicate that any instance or class hierarchy (any widget in the application) will match. However, care should be taken when using loose bindings. One common mistake is to abbreviate a hierarchy too much, so that resource settings that were supposed to apply to only one widget now apply to several. This can be particularly serious (and hard to trace) when the resources being set are translation tables or constraints.

The resource name must be the string that appears in the resource name or resource class field in a resource list. This is the value of the `XtN` or `XtC` symbolic constant used in that field of the resource list.

Any entry that is not a resource specification/value pair or does not match any resource for any widget in the application or any application resource is quietly ignored (no warning message is printed). This means that a slight error in the resource specification of an entry will cause that entry to be quietly and completely ignored. It is often difficult to detect such errors.

Lines beginning with an exclamation point (!) are treated as comments. Some people have been using # instead, since it is currently supported in MIT's sample implementation of Xt. But # is not mandated in the Xt specification and therefore may be eliminated in future sample implementations from MIT or in a vendor's implementation. You are advised to use the exclamation point. (Even in the MIT implementation, # elicits warning messages from *xrdb*.)

9.2.2 Merging of Resource Files

`XtAppInitialize` constructs the resource database by consulting the following sources of resource settings, in this order:

1. A file usually in the directory */usr/lib/X11/LANG/app-defaults* (implementation dependent) on the machine running the client, where *LANG* is an optional language string (see Section 9.2.3 for details). This file is commonly called the app-defaults file, named after the directory containing it. It is intended to be written by the application developer. The name of this file is the application class (specified in the `XtAppInitialize` call), usually the same as the application name except with the first letter capitalized. If the application name begins with *x*, then the first two letters are capitalized. (This latter convention is not always followed.) The installation procedure for the application should install this file in the correct directory (or one per language, each in a different directory).
2. A file in the directory named by the shell environment variable XUSERFILESEARCHPATH, or XAPPLRESDIR with *LANG* optionally appended (see Section 9.2.3). The filename searched for is the class name of the application. If XAPPLRESDIR is not

set, the file is looked for in the user's home directory. This file is intended to be the user's, or possibly the site administrator's, customization for the particular application. The application developer can also set XAPPLRESDIR so that resource files for an application can be in the same directory as the application. (This is particularly useful when debugging the application-defaults file.) Note that the value of this variable must end with a slash (/).*

3. Resources loaded into the RESOURCE_MANAGER property of the root window by *xrdb*. Unlike any of the resource files, these resources are accessible regardless of the machine on which the client is running. Therefore, *xrdb* saves you from having to keep track of resource files on several different systems. Typically, the user arranges to have *xrdb* run automatically from *xinitrc* or its equivalent under *xdm*, the display manager. This is intended to be the method whereby the user specifies server-wide resources (to apply to all clients no matter which system they are running on).

   If the RESOURCE_MANAGER property is not set, the resource manager looks for an *.Xdefaults* file in the user's home directory. Support for *.Xdefaults* is mostly for compatibility with earlier releases of X. (See Volume Three, *X Window System User's Guide*, for more information on using *xrdb*.)

4. Next, the contents of any file (on the system running the client) specified by the shell environment variable XENVIRONMENT will be loaded. The difference between this and APPLRESDIR above is that this is a complete path name including the file name.

   If this variable is not defined, the resource manager looks for a file named *.Xdefaults-**hostname*** (with a hyphen) in the user's home directory, where ***hostname*** is the name of the host where the client is running. (On systems with a network file system, the home directory may be on a system different from the one where the application is running, the one where the server is running, or both.)

5. Any values specified on the command line with the *-xrm* option will be loaded for that instance of the program.

6. If the application has defined any command-line options by passing an options table to `XtAppInitialize`, values from the command line will override those specified by any other resource settings.

The order in which these various sources are loaded, as shown in the list above, is the reverse order of their priority. That is, those that are loaded first will be overridden by those loaded later if an identical specification is found.

If a resource value is hardcoded in the arguments of the call to create a widget, that value takes precedence over any value for that resource in the resource database. If a widget is created and no setting exists in the database for a particular resource, the value pointed to by the `default_address` field of the resource list in the widget is used. This is also true for application resources and subresources.

*If you break the conventions by giving your program binary a capitalized name, or by giving your application-defaults file a lower-case name, it is possible to have the binary accidentally interpreted as an application default file. The entire binary will be searched for resource lines! Since the binary is so large, your application will start up *very* slowly. No error will be reported because the resource manager quietly ignores entries it doesn't understand.

Figure 9-1 shows where Xt looks for resource files and in what order, on most UNIX-based systems. The exact directories are operating system and implementation dependent. Remember that the app-defaults file is written by the application writer, and all the rest of the resource sources are for the user. In practice, few users use more than one or two of these sources of resource settings.

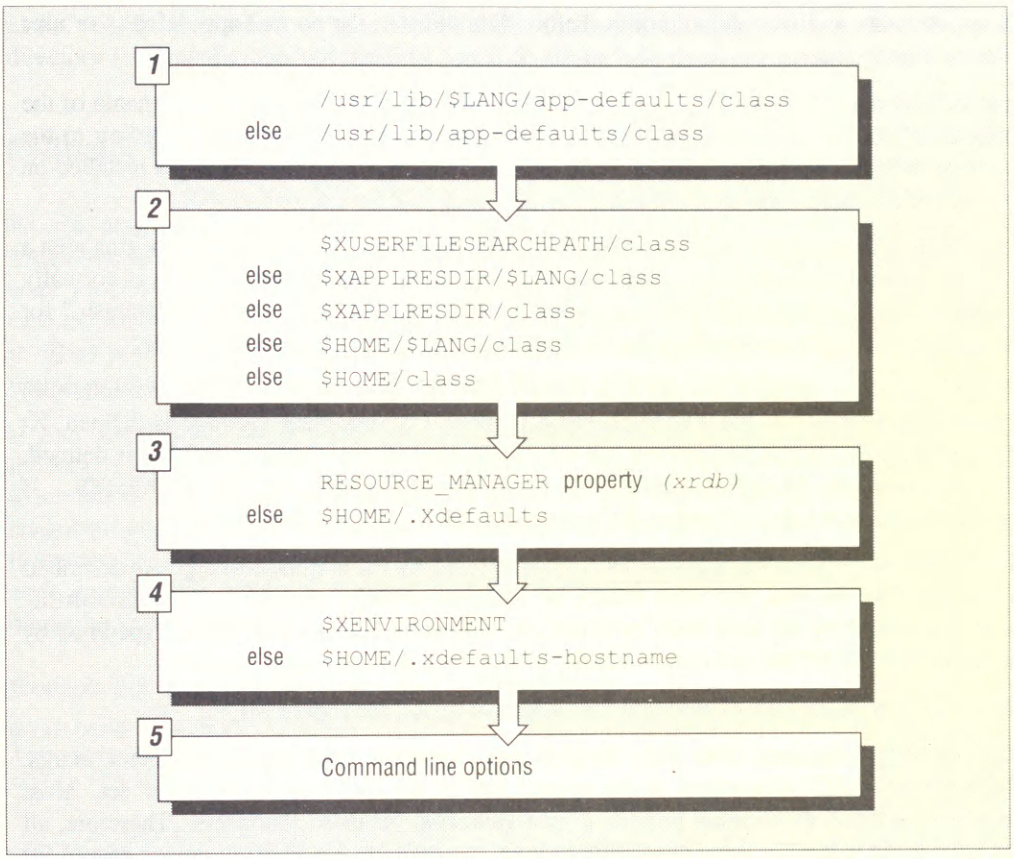


Figure 9-1. Suggested resource setting search path on UNIX-based systems

9.2.3 The Language String

In R4, Xt's resource handling was modified to support resource files for different languages. The goal is to have all the language dependencies of an application in files, so that just by

selecting a different set of resource files at runtime, the application will operate correctly in another language.*

The language to use is selected with a *language string*, which is an application resource defined by Xt: `XtNxnlLanguage`. The default language string is `NULL`, which makes Xt behave as it did in R3. But when `XtNxnlLanguage`, or the LANG environment variable, is set to a language name such as "spanish," Xt looks first in certain directories for the Spanish app-defaults and user defaults files, before defaulting to the normal app-defaults or user defaults directories.

Note that all app-defaults and user defaults files have the same name (the class name of the application); only the directory in which they are first looked for changes according to the language string. Therefore it is possible to have all the files for all the languages installed on a system at the same time.

The search path for resource files is implementation dependent, but it is always begins with a language-string-based directory. Each app-defaults file on UNIX-based systems is normally in a directory */usr/lib/X11/$LANG/app-defaults*. When the user sets LANG to "spanish," for example, Xt looks for an app-defaults file first in */usr/lib/X11/spanish/app-defaults*.

The path searched for user defaults files (which have the same name as the app-defaults file) is more complicated. If the XUSERFILESEARCHPATH environment variable is defined, Xt follows that path. If it is not defined but the XAPPLRESDIR environment variable is defined, Xt will look in $XAPPLRESDIR/LANG first before looking in $XAPPLRESDIR. If XAPPLRESDIR is not defined either, Xt looks in $HOME/LANG and then $HOME.

The remaining resource setting sources are not affected by the language string. As described in the previous section, the next source of resource settings merged is the RESOURCE_MANAGER property set with *xrdb*, or if not set, *.Xdefaults*, followed by the file specified by the XENVIRONMENT environment variable, or if not set, the *.Xdefaults-host* file.

For a summary of the path of resource files searched by Xt, see Figure 9-1.

Most western languages other than English use characters with accents or other marks. These are non-ASCII characters, in the second half of the ISO Latin-1 character set. Most fonts now in the X distribution provide glyphs (bitmaps) for these characters. Therefore, all you have to do is specify these characters in your app-defaults file in the directory named for the desired language. However, most standard English keyboards do not provide a way to type non-ASCII characters (and some computers can't store them). Because of this, the resource file format has been augmented to provide a special syntax for specifying non-ASCII characters. Each character is specified using a backslash followed by a three-character octal number that represents the character. You can use *xfd* to determine the proper octal number to use. Just click on the desired character to have its index displayed. This index is the number you enter in the app-defaults file. For example, Example 9-2 shows a Spanish app-defaults for *xhello*.

---

*It is important to note that no one has actually done this yet! The mechanism is too new. In fact, the X Consortium is currently reviewing the support for internationalization in Xt and Xlib to determine whether they are sufficient to acheive the above goal. Most likely they will recommend some additional features, but of course since both Xt and Xlib are Consortium standards, all changes must be backwards compatible.

Example 9-2. An app-defaults file for the Spanish language

```
*hello.font:      *courier-bold*18*iso8859-1
*hello.label:     \161Hola, Mundo!
```

The sequence 161 produces the inverted exclamation point that begins the Spanish exclamatory sentence. It is important to know for sure that the font you choose has the desired glyphs. Note that `iso8859-1` means ISO Latin-1, so that if you specify it in your font name, you are guaranteed to have all ISO-Latin-1 characters. Many of the fonts in the *misc* directory (such as the standard terminal fonts) do not include these characters.

9.2.4 Fallback Resources

As you may recall from chapters 2, 3, and 4, `XtAppInitialize` has an argument in which you can specify fallback resources, which we didn't use. Many applications won't operate properly without their application defaults file. *Fallback resources* are a defense in case the user doesn't install the app-defaults file, or if something happens to prevent access to it. They provide minimal application-specific resource settings that either allow the application to run safely or instruct the user to install the app-defaults file properly.

If your app-defaults file is small, you can (and should) put the whole app-defaults file in the fallback resources. However, if your app-defaults is large, it probably makes more sense just to set one or more labels in the application to tell the user that the app-defaults file is not installed properly or cannot be accessed. Remember that the fallback resources are used only if the app-defaults file is not found. Fallback resources are a `NULL`-terminated list of strings, each containing a resource setting. Example 9-3 shows how they are declared and then passed to `XtVaAppInitialize`.

Example 9-3. Setting fallback resources in XtAppInitialize

```
main(argc, argv)
int argc;
char **argv;
{
     XtAppContext app_context;
     Widget topLevel, hello;

     static String fallback_resources[] = {
         "*hello.label: App-defaults file not installed or not\
             accessible.",
         "*hello.font: *courier-bold*18*iso8859-1",
         NULL,          /* Must be NULL terminated */
     };
     topLevel = XtVaAppInitialize(
             &app_context,          /* Application context */
             "NoFile",              /* Application class */
             NULL, 0,               /* command line option list */
             &argc, argv,           /* command line args */
             fallback_resources,    /* for missing app-defaults file */
             NULL);                 /* terminate varargs list */
     .
     .
}
```

Note that there is also a separate function, `XtAppSetFallbackResources`, that can be used to set the fallback resources separately from the `XtAppInitialize` call.

9.2.5 Resource Matching Algorithm

When a widget is created, its expanded instance hierarchy and class hierarchy together with its resource names and classes are compared to each entry in the merged resource database. To demonstrate how matches are made, we'll look at a sample widget hierarchy and follow the process of finding the value for one resource of one widget from the merged resource database. Figure 9-2 shows the widget instance hierarchy for the `quit` widget in the *xbox* application shown in Chapter 3, *More Techniques for Using Widgets*. The figure also shows the corresponding fully specified instance and class names for the `quit` widget. This section describes how this widget's resources are set by the resource manager.*

We know that `quit` is a Command class widget and therefore that Xt will be searching the resource database for each resource in Command's resource list (and the resources in its superclasses' resource lists). It will search for one resource at a time. To demonstrate the conflicts that can occur, we'll use the Core resource `XtNbackground`, which is common to all widgets. It will appear in the resource database as `background`.

The matching process can be thought of as a process of elimination. Let's assume the merged resource database is as shown in Example 9-4.

Example 9-4. A sample merged resource database

```
*box.background:                blue        (entry 1)
*background:                    red         (entry 2)
*quit.background:               green       (entry 3)
*quit.label:                    Quit        (entry 4)
*Command.background:            yellow      (entry 5)
*Box.Command.background:        violet      (entry 6)
*box*background:                pink        (entry 7)
xbox.background:                orange      (entry 8)
```

Only resource database entries that specify `background` as the last element before the colon are possible matches. That eliminates entry 4. The fully specified instance and class hierarchies are then compared with each possible match, beginning with the first component in each hierarchy.

*The actual algorithm used by Xt differs slightly from that described here, because there are shortcuts that the resource manager takes that are hard to follow even if you have the source code. However, the algorithm described here gives the same result, with more clarity.

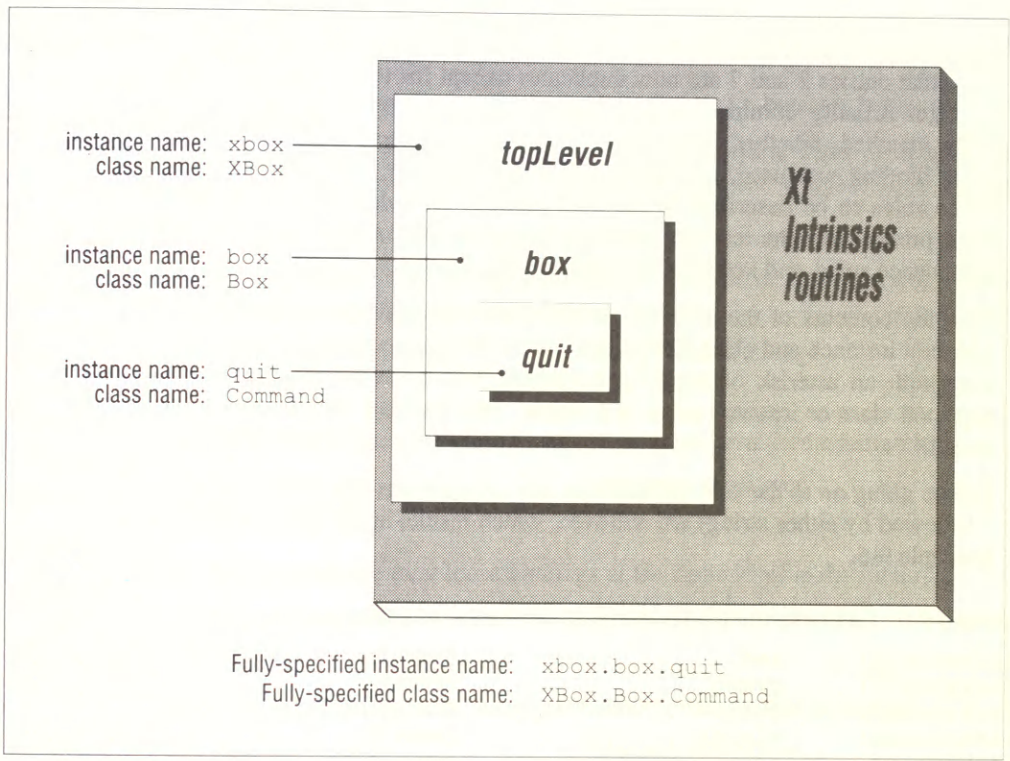


Figure 9-2. The quit widget in a sample widget hierarchy

1. Every entry beginning with the asterisk wildcard binding (*) as well as the one beginning with `xbox`, matches `xbox`, the first component of the fully specified instance name. All those beginning with * also match the first component of the fully specified class name, `XBox`. Since entry 8 actually contains the string `xbox`, the `xbox` component is removed for comparison at the next level. Entry 8 now begins with `.background`.

2. The first component of each resource specification (after removal of previously matched components) is now compared to the second element in the widget's class and instance hierarchies. This should be either `box` or `Box`. All the entries that begin with * still match, because * matches any number of levels in the hierarchy. However, there is no second element in entry 8, once the resource name `background` is removed. Therefore, entry 8 is eliminated. Also, since entries 1, 6, and 7 actually contain the strings `box` or

son of the next level. Example 9-5 shows the resource database as it would appear after the components and entries eliminated so far.

Example 9-5. Sample resource database with eliminated entries and components

```
.background:                blue        (entry 1)
*background:                red         (entry 2)
*quit.background:           green       (entry 3)
*Command.background:        yellow      (entry 5)
.Command.background:        violet      (entry 6)
*background:                pink        (entry 7)
```

Note that entries 2 and 7 are now duplicates except for the resource value. The resource manager actually eliminates one of these entries based upon the levels at which each entry matched, whether the instance name or class name matched, whether a tight or loose binding was used, and which had more elements specified. These are the precedence rules to be described in the next section. In order to keep the example clearer, we'll pretend that the resource manager keeps the information necessary to apply all the precedence rules, and keeps all the entries, until the end.

3. Now the contents of the resource database are compared to the third component in the widget's instance and class hierarchies, `quit` and `Command`. As usual, anything beginning with an asterisk or anything beginning with a period (.) followed by either the expected class or instance name is a match. This matches all but entry 1, which is eliminated.

   Before going on to the next comparison, any components that matched specifically (a . or * followed by either string) are removed, which results in the resource database shown in Example 9-6.

Example 9-6. Resource database after final elimination of entries and components

```
*background:      red         (entry 2)
.background:      green       (entry 3)
.background:      yellow      (entry 5)
.background:      violet      (entry 6)
*background:      pink        (entry 7)
```

Now you see that we are left with only the resource names and tight or loose bindings. The matching process is finished, and the precedence analysis begins. The next section describes the precedence rules and then finishes this example to determine the priority of the finalist entries.

9.2.6 Resource Precedence Rules

Because of the way merging works, no two resource specifications in the merged resource database will be alike. (Remember that we are using the term *specification* for the part of the resource setting up to and including the colon.) For example, the merged database could never contain both of the following:

```
XBitmap*box*background: green
XBitmap*box*background: red
```

because the merging process would remove the setting that appeared earlier in the list of database sources.

However, the database could contain two or more resource settings that apply to the same resource of the same widget, because of differences in the widget class or instance hierarchy or the bindings. For example, the database could contain:

```
XBitmap*box*background: green
XBitmap*quit.background: red
```

If the `quit` button is a child of `box`, both settings apply to the `quit` button's background.

The resource manager provides a set of rules that govern which setting takes precedence in cases where there are two settings for the same resource of the same widget. Here are the four rules:

1. A specification that includes higher components in the instance or class hierarchy takes precedence over one that includes only lower ones.

   ```
   *topLevel*quit.background:        takes precedence over
   *box*quit.background:
   ```

2. Instance names take precedence over class names at the same level in the hierarchy.

   ```
   *quit.background:                 takes precedence over
   *Command.background:
   ```

3. Tight bindings take precedence over loose bindings at the same level in the hierarchy.

   ```
   *box.background:                  takes precedence over
   *box*background:
   ```

4. A name or class that is explicitly stated takes precedence over one that is omitted.

   ```
   *box*quit.background:             takes precedence over
   *box*background:
   ```

To understand the application of these rules, let's return to our extended example. In the course of developing that example, we eliminated information about the level at which components occurred. However, the actual process of matching applies the precedence rules at each step. As a result, let's start again with the original appearance of the entries that pass the matching test. The remaining five as they appeared originally are shown in Example 9-7.

Example 9-7. Resource database finalists in original form

```
*background:                  red          (entry 2)
*quit.background:             green        (entry 3)
*Command.background:          yellow       (entry 5)
*Box.Command.background:      violet       (entry 6)
*box*background:              pink         (entry 7)
```

From here on, we will determine not only which one of these five will take effect, but the actual precedence of the five. In other words, once the one with highest precedence is determined, we'll see which would take effect if that one was commented out, and so on.

The precedence rules are applied in order to determine the order of the finalist entries.

1. Rule 1 specifies that a specification that contains higher components in the instance or class hierarchy takes precedence over one that contains only lower ones. The highest components that appear in our example are `box` and `Box` in entries 6 and 7. Therefore, these two have higher priority than any others.

2. To choose between these two, we continue to Rule 2. Instance names (`box`) take precedence over class names (`Box`). Therefore, entry 7 has the highest precedence, followed by entry 6. Note that the precedence comparison of two finalists proceeds in the same manner as the original matching—from left to right in the entry, one component at a time.

3. To determine the precedence of the remaining three entries, 2, 3, and 5, we begin again with Rule 1. However, Rule 1 does not apply because no two entries here specify different levels in the hierarchy. Entries 3 and 5 contain the `quit` level and entry 2 nothing (an asterisk does not count for Rule 1 because it is not a *specified* level—it is any level). Rule 2 specifies that the instance name `quit` takes precedence over the class name `Command`, and therefore entry 3 has higher priority than entry 5. Rule 3 does not apply, because no two entries are identical except for binding. Because of Rule 4 we know that both entries 3 and 5 are higher priority than entry 2, because 3 and 5 state a name or class that is omitted in 2.

Therefore, the final precedence is as shown here:

```
1.   *box*background:                pink     (entry 7)
2.   *Box.Command.background:        violet   (entry 6)
3.   *quit.background:               green    (entry 3)
4.   *Command.background:            yellow   (entry 5)
5.   *background:                    red      (entry 2)
```

Rules 2, 3, and 4 are fairly easy to understand and apply, but many people forget or are confused by Rule 1. People get used to the fact that they can set the resources of all the children of `box` with something like entry 7, but then are shocked to find that nothing happens when they attempt to override entry 7 with entry 3—entry 3 seems more specific to them. Even the following entry (using a class name) takes precedence over entry 3 because the rule about being higher in the widget hierarchy carries more weight than the rule that instance names take precedence over class names:

```
*Box*background:          pink          (entry 7)
```

The moral of this story is that there is only one way to be sure you are setting a particular resource of a particular widget in such a way as to override all other settings that might apply to that resource: you must specify all the levels in the instance hierarchy, with tight bindings between each component. (But of course, this will not work when another resource file that is merged later also specifies the same resource and all components of the same widget with the instance hierarchy separated by tight bindings.) Since there are no messages telling you which resource specifications are actually being used, you can be tricked into thinking that

you have set resources that you actually haven't. Using only tight bindings in the application-defaults file, while more trouble initially, is probably wise in the long run.

A useful tool for figuring out the priority of resource files is the *appres* utility. You specify the class name of the application on the command line, and *appres* shows you what resource settings that application will see when run. Note, however, that this does not tell you how these resource settings will actually apply to the widgets in the application, since *appres* has no knowledge of the widget instance hierarchy in your application.

Another tactic in tracing resource settings is to build a routine into your application that gets and prints all the critical resources of a widget. Using `XtGetResourceList` you can get the list of resources supported by a widget class. Then you can query each of those resources and print most of them out. Remember that certain resources are compiled into internal forms, so you can't print out translation tables, accelerator tables, or callback lists. It may also be difficult to interpret other values, since many of them will have already been converted from the string form in the resource file into the most convenient internal form. For example, colors will have been changed from strings such as "red" into a pixel value (a number which could change each time the application is run).

Figure 9-3 illustrates the entire process of resource matching.

9.3 Type Conversion

You already know that Xt is capable of converting resource values from the string representation specified in resource files to the actual type required in code. In fact, Xt does so automatically if the resource list is properly written. This section describes this process in more detail, and tells you how to create converters for converting your own data types.

In R4, a new set of interfaces for type converters was introduced. The new interfaces support display-specific conversions and better control of converted-value caching. This section describes these new interfaces, but does not describe the older ones. Be aware that many existing widgets and applications still use the older interfaces.

9.3.1 Conversions from XtRString

The primary purpose of converters is to allow resource files to contain string values for non-string program variables. (They are also used internally to convert default values that cannot be specified easily in the desired data type.) Secondly, converters confine the details of data conversion to a single routine that is registered with Xt. This is a big benefit because users of the converted types need not know the details of how the conversion takes place, or the internal definition of the target type.

Xt provides converters from `XtRString` to the representation types listed in Table 9-2. You can use these representation types as target types in resource lists in applications or widgets, without having to register them, if you want the user to be able to specify the resource in a resource file.

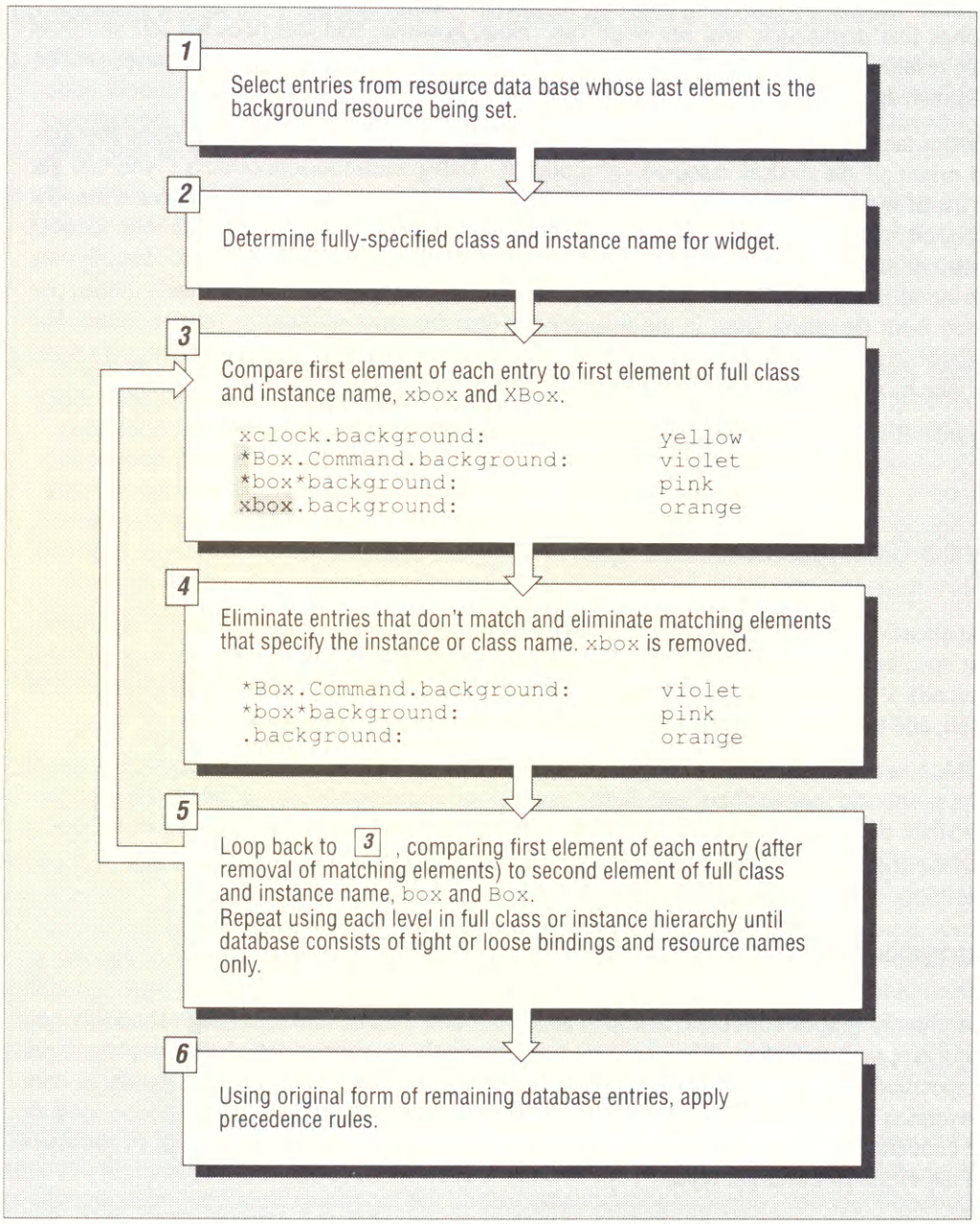


Figure 9-3. Steps in matching resource entries for one resource of widget being created

Table 9-2. Built-in Type Converters from XtRString

Target Type	Description of Converter
`XtRAcceleratorTable`	Compiles a string accelerator table into internal accelerator format (no need to call `XtParseAcceleratorTable`).
`XtRAtom`	Converts a string property name into the corresponding Atom.
`XtRBoolean`	Converts strings "true," "false," "yes," "no," "on," "off" to corresponding Boolean value (case insensitive).
`XtRBool`	Same as for `XtRBoolean`.
`XtRCursor`	Given a standard X cursor name, returns a cursor ID.
`XtRDimension`	Converts a width or height value to a `Dimension`.
`XtRDisplay`	Given a display name, opens the display and returns a `Display` structure.
`XtRFile`	Given a filename, opens the file and returns the file descriptor.
`XtRFloat`	Converts a numeric string to floating point
`XtRFont`	Given a font name, loads the font (if it is not already loaded), and returns the font ID. See Appendix B, *Specifying Fonts and Colors*, for more information on legal values. The value `XtDefaultFont` will return the default font for the screen.
`XtRFontStruct`	Given a font name, loads the font (if it is not already loaded), and returns a pointer to the `FontStruct` containing font metrics. The value `XtDefaultFont` will return the default font for the screen.
`XtRGeometry`	Given a standard geometry string, this converter simply copies a pointer to the string and calls it a new resource type.
`XtRInitialState`	Converts strings "Normal" or "Iconic" into the symbols `NormalState` or `IconicState`.
`XtRInt`	Converts a numeric string to an integer.
`XtRPixel`	Converts a color name string (e.g., "red" or "#FF0000") into the pixel value that will produce the closest color possible on the hardware. See Appendix B, *Specifying Fonts and Colors*, for more information on legal values. The two values `XtDefaultBackground` and `XtDefaultForeground` are always guaranteed to exist, and to contrast, on any server.
`XtRPosition`	Converts an `x` or `y` value to a `Position`.
`XtRShort`	Converts a numeric string to a short integer.
`XtRTranslationTable`	Compiles string translation table into internal translation table format (no need to call `XtParseTranslationTable`).
`XtRUnsignedChar`	Converts a string to an unsigned char.
`XtRVisual`	Converts a string specifying a visual class to a pointer to a supported visual structure of that class.

If there is no converter from `XtRString` to a particular resource type, it may not be possible to specify that resource type in a resource file. For example, there is no converter for `XtRCallback` since it would be meaningless to specify a function in a resource file. The

proper way to set a callback resource is with XtAddCallback or a static callback list declared in the application.

Some converters that are widely needed, however, are not provided by Xt. For example, many applications need a converter from a filename (string) to a bitmap suitable for use as an icon pixmap. Fortunately, the Xmu library contains several commonly used converters. Even though Xmu (like Xaw) is not part of the X Consortium standard, it is part of MIT's core distribution and is available on most systems.

However, because the Xmu converters are not built into Xt, you need to register them with a call to XtSetTypeConverter (from widget code) or XtAppSetTypeConverter (from application code) before using them in an application or widget.* (We'll describe the converters and show how to register them in Section 9.3.4.)

9.3.2 Other Built-in Type Conversions

While the conversions from XtRString are the most widely used, because they allow a resource to be specified from a resource file, there are also a number of built-in converters between other data types, for use internally within Toolkit programs.

Most commonly, these converters are used to convert between the resource_type and default_type fields of a resource definition.

Table 9-3 lists those converters automatically recognized by Xt.

Table 9-3. Other Built-in Converters

From	To	Description of Converter
XtRColor	XtRPixel	Converts an XColor structure to a pixel value.
XtRPixel	XtRColor	Converts a pixel value to an XColor structure.
XtRInt	XtRBoolean	Converts an int to a Boolean.
	XtRBool	Converts an int to a Boolean.
	XtRColor	Converts an int to an XColor.
	XtRDimension	Converts an int to a Dimension.
	XtRFloat	Converts an int to a float.
	XtRFont	Converts an int to a Font.
	XtRPixel	Converts an int to a pixel value.
	XtRPixmap	Converts an int to a Pixmap.
	XtRPosition	Converts an int to a Position.
	XtRShort	Converts an int to a short.
	XtRUnsignedChar	Converts an int to an unsigned char.

*Prior to R4, the calls to register type converters were XtAddConverter and XtAppAddConverter. The new interfaces provide better control of caching, and display-specific conversion.

For example, the Core resource `XtNborderPixmap` has its default value set as shown in Example 9-8.

Example 9-8. A resource definition converting an int to a pixmap

```
static XtResource resources[] = {
    .
    .
    .
     {
    XtNborderPixmap,
    XtCPixmap,
    XtRPixmap,
    sizeof(Pixmap),
    XtOffsetOf(CoreRec, core.border_pixmap),
    XtRImmediate,
    (XtPointer) XtUnspecifiedPixmap
     },
    .
    .
    .
}
```

The specified default value `XtUnspecifiedPixmap` is an integer defined to have a value that does not equal the constant `CopyFromParent` or any valid Pixmap ID. The `initialize` method for the Core widget class checks for this value, and does not set the `background` window attribute unless the application or a resource file has set the `XtNborderPixmap` resource to some value other than the default.

9.3.3 Special Resource Defaults That Do Not Use Conversion

There are two special values, `XtRImmediate` and `XtRCallProc` that can be used only in the `default_type` field of a resource definition. These values require no type conversion. The value provided in the `default_address` field must be of the correct type.

The type `XtRImmediate` means that the value in the `default_address` field is the default value itself, not its address. For a string, this means that the `default_address` field is a pointer to the string, not a pointer to a pointer.

In Example 9-9, the value in the `default_address` field of the `XtNheight` resource definition is the actual default—in this case, zero.

Example 9-9. A resource definition using XtRImmediate

```
static XtResource resources[] = {
    .
    .
    .
     {
    XtNheight,
    XtCHeight,
    XtRDimension,
    sizeof(Dimension),
    XtOffsetOf(RectObjRec, rectangle.height),
```

Example 9-9. A resource definition using XtRImmediate (continued)

```
    XtRImmediate,
    (XtPointer) 0
     },
    .
    .
    .
};
```

The type `XtRCallProc` means that the value in the `default_address` field is a pointer to a function. This function is of type `XtResourceDefaultProc`, and it is expected to retrieve the desired default value at run time. When the widget instance is created, the function is automatically invoked with these parameters: the widget ID, the resource offset, and a pointer to the `XrmValue` in which to store the result. The function should fill in the `addr` field of the `XrmValue` with a pointer to the default data in its correct type.

In Example 9-10, the value in the `default_address` field of the `XtNscreen` resource definition is the name of a function that will retrieve the screen on which the widget is displayed.

Example 9-10. A resource definition using XtRCallProc

```
static XtResource resources[] = {
    .
    .
    .
     {
    XtNscreen,
    XtCScreen,
    XtRPointer,
    sizeof(int),
    XtOffsetOf(CoreRec, core.screen),
    XtRCallProc,
    (XtPointer) XtCopyScreen
     },
    .
    .
    .
};
```

Example 9-11 shows an example of an `XtResourceDefaultProc`.

Example 9-11. An example of an XtResourceDefaultProc

```
/*ARGSUSED*/
void XtCopyScreen(widget, offset, value)
     Widget         widget;
     int            offset;
     XrmValue      *value;
{
     value->addr = (XtPointer)(&widget->core.screen);
}
```

9.3.4 Registering Type Converters

As noted earlier, not every resource type symbol defined in *StringDefs.h* is supported by a built-in converter, though the Xmu library does provide some of the most important converters that are missing in Xt. In addition, you can define your own resource types, and write converter routines to convert from a string representation in a resource file to the appropriate data type. (You can write converters from any type to any other type, but converters from String are by far the most useful.)

Table 9-4 lists the converters from `XtRString` provided by the Xmu library.

Table 9-4. Xmu Converters

From	To	Description of Converter
`XtRString`	`XtRBackingStore`	The `XmuCvtStringToBackingStore` converter converts the strings "NotUseful," "WhenMapped," and "Always" (in any case) into the corresponding constants (in proper case) for use in setting the `backing_store` window attribute. (See Volume One, *Xlib Programming Manual*, for details on backing store.)
	`XtRBitmap`	The `XmuCvtStringToBitmap` converter takes the string filename of a file in standard X11 bitmap format and creates a one-plane pixmap containing that bitmap data.
	`XtRCursor`	The `XmuCvtStringToCursor` converter converts one of the standard cursor names (from *<X11/cursorfont.h>*), a font name and glyph index of the form "FONT fontname index [[font] index]", or a bitmap file name as in `XtRPixmap` below, and converts it to an X `Cursor`.
	`XtRJustify`	The `XmuCvtStringToJustify` converter converts the strings "right," "left," or "center," in any case, to an enumeration constant suitable for use by a justify resource. This converter is used by the Athena Label widget.
	`XtRLong`	Converts a string to a long integer.
	`XtROrientation`	The `XmuCvtStringToOrientation` converter converts the strings "horizontal," or "vertical," in any case, to an enumeration constant suitable for use by an orientation resource. This converter is used by the Athena Scrollbar widget.
	`XtRShapeStyle`	Converts the strings `ShapeRectangle`, `ShapeOval`, `ShapeEllipse`, and `ShapeRoundedRectangle` in shape style constants, for use in making Xmu calls that access the Shape extension.

Table 9-4. Xmu Converters (continued)

From	To	Description of Converter
	XtRWidget	The XmuCvtStringToWidget converter converts a widget name into the corresponding widget ID. This is commonly done to specify the relative positions of the children of constraint widgets, as in the Athena Form widget resources fromHoriz and fromVert.
XtRFunction	XtRCallback	Converts a function pointer to a callback list containing that function.

Whether defined in Xmu or in your own program, a converter other than those built into Xt must be registered with a call to XtSetTypeConverter before a resource list is used that references the converted types. Resource lists are used when widgets are created or when the application calls XtGetApplicationResources. In the application, a converter must be registered after XtAppInitialize but before XtGetApplicationResources.

Within a widget, the class_initialize method is the standard place to register type converters. This method is responsible for doing processing that should be done only once when the first instance of a particular class is created.

Example 9-12 shows the code needed to register the XmuCvtStringToJustify converter in a widget. As noted above, this converter would be used for a resource (such as the Athena Label widget's label resource) designed to give the user the option of justifying text (or a graphic object) to the right, left, or center of a widget.*

Example 9-12. Registering a type converter

```
static void
ClassInitialize()
{
        .
        .
        .
    XtSetTypeConverter(XtRString,    /* source type */
            XtRJustify,              /* target type */
            XmuCvtStringToJustify,   /* converter routine */
            (XtConvertArgList) NULL,
            /* args for converter routine */
            0                        /*# args for converter routine */
            XtCacheAll,              /*  cacheing instructions */
            NULL);                   /*  destructor function */

}
```

*While it may seem a little backwards to describe how to add a converter before we say how to write one, the availability of the Xmu converters makes it likely that you would in fact want to add converters you haven't written.

Note that the non-app version of the call is used in this case, because the `class_initialize` method does not pass in a widget from which the application context could be determined.

The first two arguments of `XtSetTypeConverter` are the source and target type, respectively, specified using `XtR` symbolic constants defined in *<X11/StringDefs.h>* (or defined in an application header file or the widget public header file if the type is not standard in Xt).

The third argument is the name of the converter routine, which by convention contains the string `Cvt`. The Xmu converters all add the prefix `Xmu`.

The fourth and fifth arguments of `XtSetTypeConverter` are an argument list that will be passed to the converter routine when it is called. Some converter routines require information about the context in which the converter is called. This is usually passed in via an `XtConvertArgRec` structure, as described in the next section. If no arguments are needed, the fourth and fifth arguments of `XtSetTypeConverter` can be `NULL` and `0` respectively.

The sixth argument specifies whether the results of the conversion should be cached. The basic symbols are `XtCacheNone`, `XtCacheAll`, and `XtCacheByDisplay`. Expensive conversions should be cached. Xt caches resource values to avoid repetitive conversions of the same value, which are common in an application made up of many identical widgets. It is especially important to cache conversions that require round-trips to the server, such as color, font, and atom conversions. But it is wasteful of memory to cache silly conversions such as `XtCvtStringToGeometry`, which actually doesn't do any conversion or even validity checking. (The `XtrGeometry` representation type is a geometry string, as defined by `IXParseGeometry`. It takes no conversion to convert `XtRString` to `XtRGeometry`. The converter could at least test the string to make sure it looks like a geometry string.) Another benefit of caching is that Xt remembers unsuccesful conversions and efficiently generates warning messages without attempting any conversion more than once.

The symbol `XtCacheRefCount` can be ORed with any of the above values, in which case Xt keeps track of how many widgets still exist that used a converted value, and frees the cached value when the count reaches zero. The application, if it uses the converted value in a call to `XtGetApplicationResources`, is also counted as one reference. Reference counting is needed only if an application destroys widgets whose resource values may take up extensive space. Because reference counting takes up space and requires time, it should not be done unless necessary. To control reference counting on a widget-by-widget basis, an application must explicitly set the `XtNinitialResourcesPersistent` resource to `False` for each widget whose conversions are to be reference counted. These should be widgets that might be destroyed before the application exits.

The seventh argument of `XtSetTypeConverter` is a pointer to a procedure called a *destructor*. If the reference count for a particular resource reaches zero, Xt calls the destructor function, and removes the resource value from the conversion cache. The destructor will also be called when `XtCloseDisplay` is called if the converter was registered with `XtCachePerDisplay`. Before calling `XtCallConverter` or `XtConvertAndStore` to convert a value, the client must allocate memory in which to place the result. The job of the destructor is to deallocate this memory. If you are allowing Xt to convert data automatically, by declaring a resource list, Xt allocates the memory, and you don't need a destructor (so specify `NULL`). If you plan to call `XtCallConverter` from an application, and you

NULL for this argument. A destructor is needed only if you allocate memory dynamically for an explicit call to XtCallConverter. (Remember that reference counting happens only if the converter is registered with XtCacheRefCount set and XtNinitialResources-Persistent set to False for at least one widget. If these conditions are not met, the destructor is only called when XtCloseDisplay is called and if registered with XtCachePer-Display.)

Xt provides several functions for manipulating reference counts, which are rather obscure and mentioned here only for completeness. XtAppReleaseCacheRefs explicity decrements the reference counts for resource values converted with XtCallConverter. This would be used in the rare occasion where a value has been cached but you don't want it cached. If this is the last reference to the conversion, the registered destructor is called. Xt also defines two built-in callback functions that decrement reference counts, XtCallback-ReleaseCacheRef and XtCallbackReleaseCacheRefList.

9.3.4.1 Passing Arguments to a Type Converter

Some type converters need to be registered with additional arguments that provide information needed during the conversion. For example, XmuCvtStringToWidget needs to be passed the parent of the current widget in the application, so that it can compare the name specified in the resource to the names of the children of the parent. Example 9-13 shows the code used by a widget to register the Xmu string-to-widget converter.

Example 9-13. Adding a converter with arguments

```
static void ClassInitialize(w)
Widget w;
{
     static XtConvertArgRec parentCvtArgs[] = {
          {
               XtWidgetBaseOffset,
               (XtPointer)XtOffsetOf(CoreRec, core.parent),
               sizeof(CoreWidget)
          }
     };
          .
          .
          .
     XtSetTypeConverter(XtRString,
             XtRWidget,
             XmuCvtStringToWidget,
             parentCvtArgs,
             XtNumber(parentCvtArgs),
             XtCacheAll,
             NULL);
}
```

The format of the argument list for XtSetTypeConverter shown in Example 9-14 looks complicated, but in practice almost all converter argument lists will look very similar to the one in this example. The argument list is specified as an XtConvertArgRec:

```
typedef struct {
    XtAddressMode address_mode;
```

```
    XtPointer address_id;
    Cardinal size;
} XtConvertArgRec, *XtConvertArgList;
```

The address_mode field specifies how the address_id field should be interpreted. address_id itself is a pointer to the needed data, an offset within the widget instance structure to the data, or a function that provides the needed data. The size field specifies the length of the data in bytes.

By specifying the address mode as XtWidgetBaseOffset (see below), you can use XtOffsetOf to find the offset of appropriate data in the instance structure.* All you have to do is change the name of the instance structure field (in this case core.parent), and the structure type that contains that field (in this case WidgetRec). Notice that a pointer to this structure type appears in the sizeof call (in this case Widget).

You have just seen the normal way of specifying the converter arguments. Using different XtAddressMode values you can also specify them in other ways. The enumerated type XtAddressMode <*X11/Convert.h*>) specifies the possible values for the address_mode field:

```
typedef enum {
          /* address mode parameter representation */
         XtAddress,              /* address */
         XtBaseOffset,           /* offset */
         XtImmediate,            /* constant */
         XtResourceString,       /* resource name string */
         XtResourceQuark,        /* resource name quark */
         XtWidgetBaseOffset,     /* offset */
         XtProcedureArg          /* procedure to call */
} XtAddressMode;
```

- XtAddress causes address_id to be interpreted as the address of the data.
- XtBaseOffset causes address_id to be interpreted as the offset from the widget base address. XtWidgetBaseOffset (see below) is now slightly preferred, since it works with objects and gadgets as well as widgets.
- XtImmediate causes address_id to be interpreted as a constant.
- XtResourceString causes address_id to be interpreted as the name of a resource that is to be converted into an offset from the widget base address.
- XtResourceQuark causes address_id to be interpreted as a quark—that is, as an internal compiled form of an XtResourceString.
- XtWidgetBaseOffset is similar to XtBaseOffset except that it searches for the closest windowed ancestor if the object is not a subclass of Core. This must be used in the resource list of objects and gadgets, but can also be used for widgets.
- XtProcedureArg specifies that address_id is a pointer to a procedure to be invoked to return the conversion argument. address_id must contain the address of a function

*You can also use XtOffset instead of XtOffsetOf, but it is less portable. The biggest difference between the two is that XtOffset takes a pointer to the structure type, while XtOffsetOf takes the structure type itself. If you are converting R3 code to R4, simply change the macro name and add Rec to the end of the structure pointer type; this changes it to a structure type, if the widget uses standard naming conventions.

- `XtProcedureArg` specifies that `address_id` is a pointer to a procedure to be invoked to return the conversion argument. `address_id` must contain the address of a function of type `XtConvertArgProc`. This function type takes three arguments: an object (or widget), pointer to size (Cardinal *), and a pointer to an `XrmValue`. The value is returned in the `XrmValue`.

In most cases, you will use `XtWidgetBaseOffset` in widgets, gadgets and objects, as shown in Example 9-13.

When registering a type converter in an application rather than a widget, the structure field specified in the argument list shown in the example would be a field of the `AppData` structure instead of the instance part structure, and `CoreRec` would be replaced by `AppData`.

9.3.5 Explicitly Invoking a Converter

Converters are normally invoked by Xt because the types they convert are specified in a resource list. But this is not the only way in which converters can be invoked. It is possible to manually invoke type converters, the easiest way being `XtConvertAndStore`.* This may be useful in an application, such as for reading an icon pixmap from a file using a converter, or you may need to explicitly invoke a converter from within a converter you want to write. (Converter routines can themselves invoke other converters directly.)

One possible manual use of type converter routines is in the processing of the string parameters passed to action routines. Perhaps in the action routine itself it is more convenient to have some parameters converted to another form. For example, if an action is passed the string "True," the action code might prefer to convert this parameter to a Boolean value. The `CvtStringToBoolean` converter understands many strings that would be interpreted as Boolean, such as "Off," "On," "TRUE," "FALSE," "No," and "Yes," in upper, lower, or mixed case. It saves code to use the converter rather than comparing a string to all these strings in your own code. Example 9-14 shows an action routine of the Athena Text widget (modified for R4) in which a converter is manually invoked.

Example 9-14. Manually invoking a type converter

```
static void
DisplayCaret(w, event, params, num_params)
Widget w;
XEvent *event;              /* CrossingNotify special-cased */
String *params;             /* Off, FALSE, No, On, TRUE, Yes, etc. */
Cardinal *num_params;       /* 0, 1 or 2 */
{
      .
      .
      .
```

*`XtConvertAndStore` replaced `XtConvert` in R4. Both have the same arguments, but `XtConvertAndStore` returns `Boolean` while `XtConvert` returns `void`. `XtConvertAndStore`'s return value indicates whether the conversion succeeded or failed. `XtConvertAndStore` also implements new display-specific conversion, caching and reference counting features.

Example 9-14. Manually invoking a type converter (continued)

```
        from.size = strlen(from.addr = params[0]);
        if (XtConvertAndStore(w, XtRString, &from, XtRBoolean, &to)
                == False)
            XtAppError(XtWidgetToApplicationContext(w),
                    "DisplayCaret action: String to Boolean\
                    conversion failed");
        else {
            ; /*
               *       (*(Boolean*)to.addr) has boolean value;
               *               do something with it here */
               */
        }
    }
}
```

Note that the *from* and *to* arguments of `XtConvertAndStore` are pointers to structures containing length/pointer pairs—they are not values. The actual data is passed as a pointer to a pointer. Thus the cast to Boolean (in the comment) must dereference the pointer twice.

The widget argument of `XtConvertAndStore` is used internally by Xt as the argument for the `XtDisplay` macro, to get the pointer to the display structure, and for other purposes. In normal applications you can pass any widget here. (For some converters, the widget may need to be realized.)

`XtConvertAndStore` calls a lower-level routine called `XtCallConverter`.* If you prefer, you can use this routine. Instead of passing it the widget and the source and destination type, you pass it a display, the name of the conversion routine, any arguments to the routine, and storage in which to place the cache reference count. See Volume Five, *X Toolkit Intrinsics Reference Manual*, for details.

9.3.6 Writing a Type Converter

If your application or widget has a data type that you would like the user to be able to set through resource files, and no Xt or Xmu converter exists, you will need to write (and register) a type converter from `XtRString` to your type.

The first step in creating a converter is to decide upon the characteristics of the string you will be converting from, and the C-language type you will be converting to. Then you can copy an existing similar converter and fill in the code to convert to your desired type. Note that the R4 Athena widgets still use the R3 converter interface. To use the R4 converter functions `XtConvertAndStore` or `XtCallConverter`, or any of the cache reference counting features, you must define your converter using the R4 type `XtTypeConverter`. The difference between R4 and R3 type converters is just that R4 type converters have a display as their first argument, and they must do a display-specific conversion when a server query is involved.

*`XtCallConverter` superceded its rough equivalent R3 routine `XtDirectConvert`.

Example 9-15 shows the converter added by the Athena Form widget (changed to the R4 interface) to convert the "edge type" values for its XtNright, XtNleft, and XtNtop resources.

This is a good example of the type of converter you are most likely to write. It allows the legal values for the desired constants to be provided in either case, and uses the Xlib quark mechanism to speed string comparison. Type converters (and the Xt and Xlib resource management facilities in general) use quarks extensively to speed string comparisons. A quark is a unique ID for a string, of type XrmQuark (defined by Xlib). A call to the Xlib routine XrmStringToQuark returns the quark for a string. See Volume Two, *Xlib Reference Manual*, for details. When a nonstandard type converter that uses quarks is defined and registered in widget code, the XrmStringToQuark calls are normally placed in the class_initialize method just before the XtSetTypeConverter call.

Note that Xt manages the cache of converted values: converter routines are not responsible for caching their own returned data.

Example 9-15. An R4 XtRString to XtREdgeType type converter

```
/*  This macro or something like it used by many resource converters  */
#define done(address, type) \
          { toVal->size = sizeof(type); \
            toVal->addr = (XtPointer) address; \
            return; \
          }
    .
    .
    .
/*  Quarks used to speed string comparisons  */

static XrmQuark XtQChainLeft, XtQChainRight, XtQChainTop,
                    XtQChainBottom, XtQRubber;

/* The converter itself */
/*ARGSUSED*/
static void _CvtStringToEdgeType(display, args, num_args,
        fromVal, toVal, destructor_data)
    Display *display;               /* unused */
    XrmValuePtr args;               /* unused */
    Cardinal     *num_args;         /* unused */
    XrmValuePtr fromVal;
    XrmValuePtr toVal;
    XtPointer *destructor_data;     /* unused */
{
     static XtEdgeType edgeType;
     XrmQuark q;
     char lowerName[1000];

    /* make lower case copy */
     XmuCopyISOLatin1Lowered (lowerName, (char*)fromVal->addr);

     q = XrmStringToQuark(lowerName);

     if (q == XtQChainLeft) {
         edgeType = XtChainLeft;
         done(&edgeType, XtEdgeType);
     }
```

Example 9-15. An R4 XtRString to XtREdgeType type converter (continued)

```
    if (q == XtQChainRight) {
        edgeType = XtChainRight;
        done(&edgeType, XtEdgeType);
    }
    if (q == XtQChainTop) {
        edgeType = XtChainTop;
        done(&edgeType, XtEdgeType);
    }
    if (q == XtQChainBottom) {
        edgeType = XtChainBottom;
        done(&edgeType, XtEdgeType);
    }
    if (q == XtQRubber) {
        edgeType = XtRubber;
        done(&edgeType, XtEdgeType);
    }
    XtDisplayStringConversionWarning(display, fromVal->addr,
            "XtREdgeType");
    toVal->addr = NULL;
    toVal->size = 0;
}
```

Notice that the `XrmValuePtr` arguments passed into the converter are pointers to structures, not values. The `XrmValue` structure contains an address field and a size field.

`XtDisplayStringConversionWarning` takes as arguments a pointer to the `Display` structure, the string that could not be converted, and the type to which it could not be converted. It issues a warning message with name `conversionError`, type `string`, class `XtToolkitError`, and the default message string "Cannot convert "*`src`*" to type *`dst_type`*." (See Chapter 13, *Miscellaneous Toolkit Programming Techniques*, for more information on error and warning messages.)

9.3.6.1 Defining the Default Value

When performing conversions, such as from strings to fonts or colors, for which there is no string representation that all server implementations will necessarily recognize, a type converter should define some set of conversion values that the converter is guaranteed to succeed on, so that these can be used as resource defaults.

For example, the default string-to-pixel converter recognizes the symbols `XtDefaultForeground` and `XtDefaultBackground`. As part of its conversion, it tests for these values, and establishes the appropriate value based on the string value. The code is shown in Example 9-16.

Example 9-16. Testing for a special-case default value

```
/*
  * CompareISOLatin1 is an undocumented Xt function, allowed since
  * this converter is within Xt.  In your converters, you would use
  * XmuCompareISOLatin1.
  */
if (CompareISOLatin1(str, XtDefaultBackground) == 0) {
    *destructor_data = FALSE;
    if (pd->rv)
        done(Pixel, BlackPixelOfScreen(screen))
    else
        done(Pixel, WhitePixelOfScreen(screen));
}
if (CompareISOLatin1(str, XtDefaultForeground) == 0) {
    *destructor_data = FALSE;
    if (pd->rv)
        done(Pixel, WhitePixelOfScreen(screen))
    else
        done(Pixel, BlackPixelOfScreen(screen));
}
```

9.4 Subparts and Subresources

A subpart is a section of a widget that is replaceable but that cannot operate independently. It is just a further subdivision of the widget into smaller pieces. This section is provided only for completeness; gadgets and objects provide a more elegant way to do this.

Subresources allow subparts of a widget to have separate sets of resources. Since the Athena Text widget was the only example of the use of subresources in MIT's Release 3 Core distribution, we'll describe how Text uses subparts and subresources so you can understand the motivation behind them. You can then compare this concept with the R4 Text widget which is implemented using objects.

The R3 Text widget has three parts: the source, the sink, and the coordinator widget. The source manages the storage of data and the sink manages how it is displayed. The coordinator is the central widget that manages the communication between the source and the sink, and is inoperable without them. Both the source and the sink are replaceable pieces of code. Xaw provides only one source, which edits a string or disk file, and only one sink, which displays text in one color and in one font. The idea of providing the subparts in the first place is that they would allow enhancements to be made without changing the basic editor functionality that is in the coordinator. For example, only the source and sink would need replacing in order to implement a multifont and/or multicolor text widget.

Each subpart has its own resource list so that it truly can be replaced without any modifications to the central widget. These are the subresources.

9.4.1 The Hook Methods

The initialize_hook, set_values_hook, and get_values_hook methods are used by widgets that have subparts. They have the same function as their nonhook counterparts, except that they process only the resources of a subpart, and any subpart instance fields that depend on the subpart resources. These methods are called immediately after their nonhook counterparts.

However, the initialize_hook and set_value_hook methods have become obsolete in R4 because their arguments have been added to the initialize and set_values methods. The hook methods are still called for compatibility with existing widgets, but new widgets should move the code that would have been in the hook methods into initialize and set_values.

The get_values_hook method is passed a single copy of the widget instance structure (the *new* copy already modified in the nonhook methods), and the argument list passed to the Xt routine that triggered the method. The set_values and get_values_hook methods simply take this widget ID and argument list and pass them to XtSetSubvalues or XtGetSubvalues respectively. The initialize method uses the contents of the argument list to validate resource settings for subparts and to set nonresource subpart data.

The get_values_hook method is still used in R4. Example 9-17 shows the get_values_hook for the AsciiSrc subpart of the R3 Text widget (somewhat simplified to show the essential elements).

Example 9-17. Simplified get_values_hook method of the AsciiSrc subpart of the Text widget

```
static void
GetValuesHook(src, args, num_args)
XawTextSource src;
ArgList args;
Cardinal * num_args;
{
        .
        .
        .
    XtGetSubvalues((XtPointer) src,
            sourceResources,
            XtNumber(sourceResources),
            args,
            *num_args);
}
```

9.4.2 Managing Subresources

Managing subresources is very similar to managing application resources. Like the application, the subpart must have a structure containing the fields to set through resources. In the application you call XtGetApplicationResources or XtVaGetApplicationResources to set these fields. In a subpart, the analogous calls are XtGetSubresources or XtVaGetSubresources, which is called from the initialize

Like widgets, the resources of subparts can be queried and set manually. XtVaGetSubvalues or XtGetSubvalues queries the values, and XtVaSetSubvalues or XtSetSubvalues sets them. However, because subvalues are not part of any widget, these calls cannot identify what object is being queried or set simply by passing the widget ID. These calls have different arguments than XtSetValues and XtGetValues. Instead of the widget ID, you pass the pointer to the data structure, the resource list, and the number of resources. Therefore, XtSetSubvalues and XtGetSubvalues can be invoked only from within the widget or subpart. Actually, all these routine do is set or get the value in the specified structure.

Any application using the widget will set or get subresources using XtSetValues and XtGetValues as for normal resources, specifying only the coordinating widget as the first argument. These calls are translated into XtSetSubvalues and XtGetSubvalues calls by the set_values and get_values_hook methods. These methods are passed the arguments from the XtSetValues or XtGetValues calls and translate them into XtSetSubvalues or XtGetSubvalues calls by adding the data structure and resource list arguments. But in addition, the set_values method is responsible for validating the resource settings passed in before it calls XtSetSubvalues, and for changing any nonresource subpart structure fields like GCs that depend on resources.

10

Interclient Communications

This chapter discusses communication through the X server between an application and the window manager, and between two applications. The application-window manager communication is performed by code in the Shell widget. The application sets shell resources to control this communication with the window manager. Application-application communication is usually done with a process called selections. This form of communication is already implemented in most widgets that display text, but you may want to implement it in your own custom widgets. Selections can also pass other kinds of data such as graphics.

In This Chapter:

Window Manager Interactions 284
- Shell Subclasses 284
- Setting Shell Resources 285
- Screen Space 287
- Input Model 288
- Colormaps 290
- Icons 292
- Window Manager Decorations 293

Selections: Widget-To-Widget Communication 294
- How Atomic Selection Works 295
- Highlighting the Selected Data (Owner) 298
- Making the Selection with XtOwnSelection (Owner) 303
- Requesting the Selection (Requestor) 304
  - Possible Target Type Atoms 305
  - The Paste Action from BitmapEdit 307
- Converting the Selection (Owner) 307
- Finally Pasting the Selection (Requestor) 310
- If the Selection is Lost (Owner) 312

When the Selection Transfer is Complete (Owner) 312
ICCCM Compliance 312
Xmu Atom Caching 314
Converting the Standard Selections 315
The Clipboard Selection 316
How Incremental Selection Works 317
Miscellaneous Selection Routines 318

10
Interclient Communications

Applications share the server with other clients. Server resources, such as screen space and colormaps, must be used in a responsible, consistent manner so that applications can work effectively together. In most window systems, the window system itself embodies a set of rules for application interaction. However, the X Protocol, Xlib, and Xt were all specifically designed to avoid arbitrary conventions, so that they provide "mechanism, not policy."

Instead, the conventions covering interclient communication are described in a separate document, adopted as an X Consortium standard in July, 1989, called the *Inter-Client Communication Conventions Manual* (ICCCM). This chapter will not fully describe the ICCCM, because the job of implementing its rules is given over to a special client called the window manager and a special widget class called Shell and its subclasses.* As a result the details of the ICCCM are, for the most part, irrelevant to the application writer's needs.

In X Toolkit programs, the Shell widget returned by `XtAppInitialize` and used as the top-level window of the application automatically handles most of the required interactions with the window manager. However, the Shell widget needs additional information in certain areas. For example, the application needs to provide an icon pixmap so that the window manager can iconify it properly. The first section in this chapter describes how to set Shell resources to control how an application interacts with the window manager. This portion of the chapter is for application writers, regardless of whether you need to write widgets for your application.

In X Toolkit applications, widgets can communicate with other widgets using a mechanism called *selections*, which in turn is based on an X mechanism for common storage called *properties*. Whether the widgets involved in transferring selections are part of the same application or different applications is irrelevant. The communication between widgets takes place without input from the application. However, it can be used as a means of communication between applications. The second major section in this chapter will describe these concepts and how to implement selections between your own custom widgets. Only if your application requires a custom widget that must communicate with other widgets should you actually have to write this code. Thus, this part of the chapter is primarily for widget writers.

*If you do need to look up certain details of the ICCCM, see Appendix L, *Inter-Client Communication Conventions*, in a version of Volume Zero, *X Protocol Reference Manual*, printed on or after June, 1990 (the conventions have changed since the version of the ICCCM printed in earlier editions of Volume One). The ICCCM is also included in *troff* source form in the standard X distribution from MIT.

10.1 Window Manager Interactions

The window manager was introduced in Chapter 1, *Introduction to the X Window System*, but little mention of it has been made since then. You may recall that the window manager is just another client running on a server, except that it is given special authority to manage screen space and other limited server resources like colormaps.* To let the window manager do a better job of mediating competing demands of the various clients, each client gives the window manager information called *window manager hints*. These hints specify what resources each client would like to have, but they are only hints; the window manager is not obligated to honor them, and the client must not depend on them being honored.

Application code has little to do to interact properly with the window manager. The Shell widget returned by `XtAppInitialize` takes care of setting the essential window manager hints. However, there are a number of optional window manager hints that the application may wish to have passed to the window manager. This is done mainly by setting resources of the Shell widget. Also, there are variations in window managers and it takes some effort to make some applications work equally well under all of them.

The next few sections describe the various resources of the Shell widget, including how and when they should be set. Because the Shell widget is part of the Xt standard, these resources are present when writing applications with any widget set.

10.1.1 Shell Subclasses

There are several types of Shell widgets. The Shell widget class itself, specified by the class structure pointer `shellWidgetClass`, is never instantiated directly in applications. Only its subclasses are used. You have seen two subclasses of the Shell widget used earlier in this books: the one used for the application's top-level widget and the one used for pop ups. The application's top-level widget is created by passing the class structure pointer `applicationShellWidgetClass` as the widget class argument to `XtAppCreateShell`; this call is also made internally by `XtAppInitialize`. Pop-up shells for dialog boxes are created by passing `transientShellWidgetClass` as the widget class argument to `XtCreatePopupShell`. There are two other subclasses of Shell that are commonly used in applications. One is the `OverrideShellWidgetClass`, passed to `XtCreatePopupShell` when the shell is used for pop-up menus. The convention is this: the shell should be an OverrideShell when the pointer is grabbed to prevent other windows from getting input while the pop up is up, and the shell should be TransientShell for other pop ups. This will be discussed further in Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*.

The other additional subclass of Shell is `topLevelShellWidgetClass`, which is used to create additional, non-pop-up, top-level shells. Some applications have multiple permanent top-level windows. One of the top-level shells would be of the `applicationShellWidgetClass`, and the rest would be of the `topLevelShellWidgetClass`. Each would have a separate icon.

---

*Note that we are using the term *resources* here in a general sense, rather than implying its Xt-specific meaning.

10.1.2 Setting Shell Resources

Shell resources are primarily a way for the user and the application to send in data to be communicated to the window manager. These window manager hints control several major areas of window manager activity: they manage screen space, icons, and keyboard input. We'll discuss these areas one at a time in the following sections. Table 10-1 lists the Shell widget's resources with a brief description of what they control, and whether the application, the user, or Xt normally sets them.

As indicated in Column 3 of the table, some Shell resources are intended to be set only once. These set-once resources can be left to their default values, set in the application-defaults file, or they can be set in the code before the Shell widget is realized; but they should not be set with `XtSetValues` after realization.

Table 10-1. Shell Resources

Resource	Purpose	When Settable
usually set by Xt or Shell itself, depending on the subclass:		
`XtNargc`	Command-line args count	-
`XtNargv`	Command-line args	-
`XtNoverrideRedirect`	Set for pop-up shells not to be decorated	-
`XtNtransient`	Set for pop-up shells	-
`XtNwaitForWm`	Whether to wait at all	-
`XtNwindowGroup`	Links pop ups to main window	-
`XtNwmTimeout`	Waiting time for slow wm	-
usually set by user:		
`XtNiconX`	Icon position	Before realization
`XtNiconY`	Icon position	Before realization
`XtNiconic`	Sets `XtNinitialState` to iconic	Before realization
`XtNgeometry`	Initial size and position	Before realization
`XtNtitle`	String for title bar	Anytime
usually set by application:		
`XtNallowShellResize`	Does shell ask wm for size change?	Anytime
`XtNbaseHeight`	Height of fixed components	Anytime
`XtNbasewidth`	Width of fixed components	Anytime
`XtNheightInc`	Desired height increment	Anytime
`XtNwidthInc`	Desired width increment	Anytime
`XtNiconMask`	Mask used with icon pixmap	Before realization
`XtNiconName`	String for icon	Before realization
`XtNiconPixmap`	Picture for icon	Before realization
`XtNiconWindow`	Window for icon	Before realization
`XtNinitialState`	Whether normal or iconic	Before realization
`XtNinput`	Keyboard input model	Before realization
`XtNmaxAspectX`	Maximum aspect ratio x/y	Anytime
`XtNmaxAspectY`	Maximum aspect ratio x/y	Anytime
`XtNmaxHeight`	Maximum acceptable height	Anytime

Table 10-1. Shell Resources (continued)

Resource	Purpose	When Settable
XtNmaxWidth	Maximum acceptable width	Anytime
XtNminAspectX	Minimum aspect ratio x/y	Anytime
XtNminAspectY	Minimum aspect ratio x/y	Anytime
XtNminHeight	Minimum acceptable height	Anytime
XtNminWidth	Minimum acceptable width	Anytime
XtNsaveUnder	Should server save under when mapped	Before realization

Several of the Shell resources are set automatically by Xt or the window manager and under normal circumstances should not be modified by an application:

- The XtNargc and XtNargv resources are set by Xt to contain the command-line arguments used to invoke the application. These values may be used by the window manager or a session manager to allow the user to reinvoke the application using the same arguments.
- The XtNwindowGroup resource is used to link pop-up windows to an application's main window. If not set explicitly, XtNwindowGroup is automatically set to the top-level ancestor of the pop-up shell, which is usually the application's top-level shell.
- The XtNtransient and XtNoverrideRedirect resources are set automatically by Xt depending on what kind of Shell widget you create. For top-level application shells, both are set to indicate that the window manager should treat this window as a top-level window. For TransientShell pop-up shells, XtNtransient is set automatically to the shell specified in XtNwindowGroup. When the top-level window is iconified, the TransientShell pop up will also be iconified. The TransientShell may also be decorated differently from main application shells.
- XtNoverrideRedirect defaults to TRUE in the OverrideShell class, indicating that the pop up can map itself completely without window manager intervention. This type of pop-up shell is typically used for pop-up menus, because they should not be decorated or interfered with by the window manager.
- The XtNwaitForWm and XtNwmTimeout resources control the response to delays by the window manager in responding to geometry change requests. By default, XtNwaitForWm is TRUE, and XtNwm_timeout is five seconds. When making a geometry request to the window manager, the Shell widget will wait for five seconds for a response. If a response does not arrive within five seconds, the Shell widget will set XtNwaitForWm to FALSE, assume that the window manager is not functioning, and continue, without waiting for the event that should arrive to represent the size of the window, and without updating Xt's internal cache of window geometries. When this event does arrive later, Xt may set XtNwaitForWm back to TRUE and update its internal cache. These resources should normally be left to their default values.

The XtNgeometry and XtNiconic resources are intended to be specified by the user. Only these two resources have standard command-line options for setting them. The XtNgeometry resource is settable using a command-line option of the form:

```
-geometry [width{x}height][{+-}xposition{+-}yposition]
```

(Either the size or position portion of the geometry string may be omitted, as indicated by the square brackets. In specifying the *x* or *y* position, a positive value, indicated by a plus sign, is relative to the top or left side of the reference window, while a negative value, indicated by a minus sign, is relative to the bottom or right side.)

The *-iconic* command-line option, if present, sets the `XtNiconic` resource, indicating to the window manager that the application should start up as an icon.

The `XtNicon_x` and `XtNicon_y` icon position hints are best left to be specified by the user. Icons' positions are determined by the window manager based on these hints or on an icon-positioning policy. An application with several top-level windows could set the icon position hints in its application-defaults file so that the icons for each top-level window appear side-by-side.

We will discuss the remaining resources in related groups. Section 10.1.3 discusses the ones related to the size of the application's main window and other screen space issues. Section 10.1.4 describes the keyboard input model hint. Section 10.1.5 describes how applications should handle colormaps to cooperate with the window manager. Section 10.1.6 describes the ones that apply to the application's icon.

10.1.3 Screen Space

The Shell widgets of each application have windows that are children of the root window. These are called top-level windows. The window manager directly controls the size and position of the top-level windows of each application. The window manager does not control the geometry of other widgets that are the descendants of the Shells, except indirectly through the geometry management mechanism described in Chapter 11, *Geometry Management*.

The most basic size hint, the one that specifies simply the desired initial height and width, is automatically set by Xt based on the size of the child of the Shell widget. This hint is used by most window managers to display the outline of your application when it first appears on the screen, ready for the user to place and/or size the application. If your application does not have specific size needs, you need not set any additional resources.

The additional size hints specify the application's range of desired sizes, desired increments of sizes, and desired range of aspect ratios. These are set by the `XtNbaseHeight`, `XtNbaseWidth`, `XtNminWidth`, `XtNminHeight`, `XtNmaxWidth`, `XtNmaxHeight`, `XtNwidthInc`, `XtNheightInc`, `XtNminAspectX`, `XtNminAspectY`, `XtNmaxAspectX`, and `XtNmaxAspectY` resources.

Size increment hints are useful for applications that prefer to be in units of a particular number of pixels. Window managers that listen for this hint always resize the window to the base size (for each dimension), plus or minus an integral multiple of the size increment hint for that dimension. If the base size resource has not been set, the minimum size is used as the base. For example, *xterm* uses the font width and font height as width and height increment hints, because it prefers not to have partial characters or dead space around the edges. The bitmap editor application described in Chapter 4, *An Example Application*, should

probably set both the width and height increments to the `cell_size_in_pixels`, since the bitmap cells are square.

Most applications that use size increment hints redefine the interpretation of geometry specifications (the `XtNgeometry` resource, settable through the *-geometry* standard command-line option) to reflect the size increments. For example, the width and height in *xterm* geometry specifications are in units of characters, not pixels. The Vt100 widget within *xterm* implements this by having an `XtNgeometry` resource separate from the shell geometry resource with that name. The entry for this resource in the widget resource list specifies the Vt100 instance structure field, not the shell instance structure field. Then the `realize` method for the Vt100 parses the geometry string with `XParseGeometry` (an Xlib routine that returns four separate variables containing the size and position from the geometry string) and sets the width and height fields of the core structure and the various size hints according to the returned variables. (The Vt100 within *xterm* is not a real, self-sufficient widget. For one thing, it has no `set_values` method. For any real widget to implement this approach to interpreting geometry specifications, the `set_values` method would have to multiply the `XtNwidth` and `XtNheight` resource settings by the increment hints.)

An aspect ratio is the ratio of the width measurement to the height measurement or vice versa. The `XtNminAspectX` and `XtNminAspectY` resources are used together to determine one extreme of acceptable aspect ratios, and `XtNmaxAspectX` and `XtNmaxAspectY` determine the other extreme. For example, to suggest that the *xmh* application never be more than four times larger in one direction than it is in the other, the following values (as they would appear in the application-defaults file) would suffice:

```
xmh*topLevel.minAspectX:    1
xmh*topLevel.minAspectY:    1
xmh*topLevel.maxAspectX:    4
xmh*topLevel.maxAspectY:    4
```

Remember that every application must be able to do something reasonable given any size for its top-level window, even if the window is too small to be useful or if any of these hints are ignored by the window manager.

10.1.4 Input Model

Window managers also control which window keyboard input will be delivered to by setting the *keyboard focus window* (sometimes called just the *keyboard focus*) to an application's top-level window. The distribution of events according to the current keyboard focus is handled by the server; it is a basic feature of X, not of Xt. Some window managers, like *twm*, use the pointer-following (also called real-estate-driven) model of keyboard input; the window containing the pointer gets the keyboard input. Setting the keyboard focus once to the root window implements the pointer-following model. Other window managers use the click-to-type model, exemplified by the Macintosh™ user interface, requiring that the user click the pointer in the window where keyboard input is to go. The window manager sets the keyboard focus to the window the user clicks on. Some window managers let the user select between these two focus models, either at startup or at any time. However, these basic window manager behaviors can be modified by the input model hint, which is set by the `XtNinput` resource.

If XtNinput is set to TRUE, the window manager will set the keyboard focus to this application or not, according to its pointer-following or click-to-type model of keyboard input. However, if it is set to FALSE, the window manager will not set the keyboard focus to this application. If the application sets this resource to FALSE and wants input, it will have to forcefully take the keyboard focus, and then put it back to the original window when finished.

For historical, not logical, reasons, the Intrinsics default for the XtNinput resource is FALSE. However, there is a special internal Shell widget class called VendorShell, which sets appropriate resources for a given widget set. The proper default for a given widget set (which usually has an accompanying window manager) is set by that widget set's VendorShell. The majority of applications need XtNinput set to TRUE unless they don't require keyboard input (and expect never to have accelerators).

There are four models of client input handling defined by the ICCCM:

- No Input. The client never expects keyboard input. *xload* is an example of such an output-only client. This type of client sets XtNinput to FALSE.
- Passive Input. The client expects keyboard input but never explicitly sets the input focus. This describes the vast majority of applications that always accept keyboard input in the window that contains the pointer. This type of client sets XtNinput to TRUE.
- Locally Active Input. The client expects keyboard input, and explicitly sets the keyboard focus, but only does so when one of its windows already has the focus. An example would be a client with subwindows defining various data entry fields that uses Next and Prev keys to move the keyboard focus between the fields, once its top-level window has received the keyboard focus. This type of client sets XtNinput to TRUE.
- Globally Active Input. The client expects keyboard input, and explicitly sets the input focus even when the focus is in windows the client does not own. An example would be a client with a scrollbar that wants to allow users to scroll the window without disturbing the keyboard focus even if it is in some other window. It wants to temporarily set and then reset the keyboard focus when the user clicks in the scrolled region, but not when the user clicks in the scrollbar itself. Thus, it wants to prevent the window manager setting the keyboard focus to any of its windows. This type of client sets XtNinput to FALSE.

Note that even if the XtNinput resource is not set to TRUE, your application will still work under some window managers, including *uwm* and the version of *twm* that is standard in Release 4. This is because these window managers use the pointer-following keyboard focus model and ignore this hint. However, it is not wise to assume that all window managers will ignore this hint. Therefore, if your application expects keyboard input, and is not of the globally active type described above, you can use the code shown in Example 10-1 to set the XtNinput resource.

Example 10-1. Setting the XtNinput resource of a Shell widget

```
main(argc, argv)
int argc;
char *argv[];
{
        .
        .
        .
    /* create the Shell widget, setting resource */
    topLevel = XtVaAppInitialize(&app_context, "Xmh", table,
                                 XtNumber(table),
            &argc, argv, NULL,
            XtNinput, (XtArgVal)TRUE,
            NULL);
        .
        .
        .
}
```

Note that the `XtNinput` resource should always be hardcoded, since the application may fail if the user is allowed to change the expected style of keyboard focus.

For a further discussion of the keyboard focus, see Section 13.3.

10.1.5 Colormaps

On most color systems, the display uses one or more hardware registers called *colormaps* to store the mapping between pixel values and actual colors, which are specified as relative intensities of red, green, and blue primaries (RGB values).

X allows virtual colormaps to be created by applications. Some high-performance systems even allow all virtual colormaps to be installed in hardware colormaps and used at the same time, even to the level of one colormap per window. Far more commonly, though, there is only one hardware colormap, and virtual colormaps have to be copied into the hardware colormap one at a time as needed. Copying a virtual colormap into the hardware colormap is called *installing* the colormap, and the reverse process where the default colormap is installed is called *uninstalling*. The window manager is responsible for installing and uninstalling colormaps.

If your application has standard color needs (decoration only), then you do not have to worry about the effects of colormaps being installed and uninstalled. Your application should use the standard `XtRString` to `XtRPixel` converter to translate color names into pixel values. If the colormap is full, or becomes full at any of the color allocations in the converter, the warning messages place the burden on the user to kill some applications in order to free some colormap cells. (If the converter cannot allocate the desired color, it prints a warning message, and the default for that resource is used. If the default is not `XtDefaultForeground` or `XtDefaultBackground`, it also must be converted and this may also fail. If both allocations fail, the color will default to black. If this is not acceptable, then you will need to write your own type converter.)

If your application absolutely requires accurate colors, a certain number of distinguishable colors, or dynamically changeable colors, you will need to write your own converter to allocate colors. For example, if a color allocation for a known correct color name string fails, it means that all the colormap entries have already been allocated and no entry for that color is available. In this case, your converter might call the Xlib call `XCopyColormapAndFree` to copy the allocations your application has already made into a new colormap. Then the converter would allocate the color (and all subsequent colors) from the new colormap.

See Chapter 7, *Color*, in Volume One, *Xlib Programming Manual*, for details of various color allocation techniques.

The window manager is responsible for installing and uninstalling colormaps according to its own policy. Typically, the policy is to install the colormap of the application that has the keyboard focus or that contains the pointer. When an application has created a new colormap on a system that supports only one hardware colormap, and that colormap is installed, all applications that were using the other colormap will be displayed using the new colormap. Since the pixel values from the old colormap have not been allocated in the new colormap, all applications that use the old colormap will be displayed with false colors. This is known as "going technicolor."

The window manager may also create standard colormaps to facilitate the sharing of colors between applications. A standard colormap is a colormap allocated with a publicly known arrangement of colors. The Xlib routine `XGetStandardColormap` allows an application to determine whether the window manager has created a specific standard colormap, and it gets a structure describing the colormap. Xmu provides numerous utilities for creating and using standard colormaps—see Volumes One and Two for a description.

If your application requires more than one colormap, each installed at the same time in a different window, you need to tell the window manager about this so that these colormaps can be properly installed when the application is active. You do this by calling `XtSetWMColormapWindows`, specifying the ApplicationShell widget as the first argument, a list of widgets that need certain colormaps as the second argument, and the length of the list as the third argument. This instructs the window manager to read the `colormap` window attribute of each of the listed widgets, and to install that colormap in each widget (if possible) whenever the application is active. (Of course, any application that depends on the existence of multiple simultaneous colormaps is doomed to run on only a small percentage of existing systems.)

The `colormap` window attribute of a window is an unchangeable characteristic of that window, assigned when the window was created (when the widget was realized). The Core `realize` method sets it based on the `XtNcolormap` Core resource of the widget. Therefore, if a widget must have a certain colormap, you must set its `XtNcolormap` resource *while creating the widget*. Attempting to set `XtNcolormap` after a widget is created will have no effect on its colormap window attribute.

10.1.6 Icons

The window manager always manages the icons for each application. Depending on the window manager, these icons may simply contain a text string called the icon name, or they may contain a pattern that identifies the application, called the icon pixmap.

Figure 10-1 shows the icon, under the *uwm* window manager, for an application with a custom icon pixmap (*xcalc*) and for one without (*xterm*).

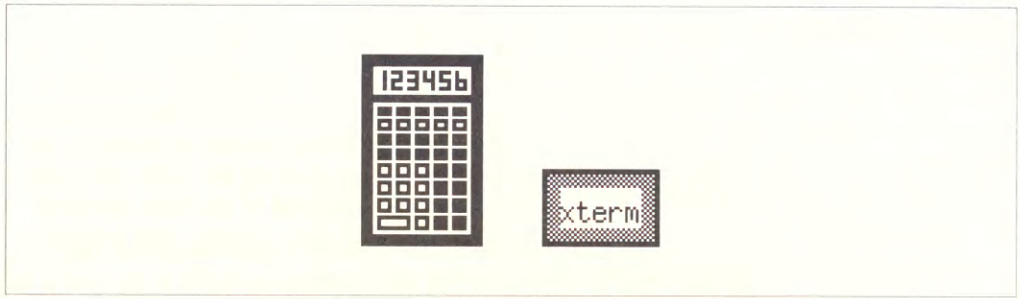


Figure 10-1. An icon with and without an icon pixmap

An application may supply an icon name by setting the `XtNiconName` resource; if it does not, the window manager will usually use the application name or the value of the `XtNtitle` resource.

The application should supply the pattern for the icon, in the form of a single-plane pixmap, as the `XtNiconPixmap` resource. There are two basic ways to do this: one is by including bitmap data, and the other is by reading it in at run-time. The former is easy to do with an Xlib call; the code that needs to be added is shown in Example 10-2. The latter technique, because it involves building a filename and looking in a number of locations for the file, is better done with a converter defined by the Xmu library. This converter technique is more complicated because the converter has to be registered and called with the proper arguments. The example using a converter is provided in the example source code for this book (*xicon2*), but is not shown here. See Chapter 9, *Resource Mangement and Type Conversion*, for more information on invoking converters.

Example 10-2. Creating an icon pixmap, and setting XtNiconPixmap

```
   .
   .
   .
/* both R3 and R4 (part of Xt) */
#include <X11/Shell.h>

#include "icon"

main(argc, argv)
int argc;
char **argv;
{
    Pixmap icon_pixmap;
```

Example 10-2. Creating an icon pixmap, and setting XtNiconPixmap (continued)

```
        .
        .
        .
    /* create topLevel here */
    icon_pixmap = XCreateBitmapFromData(XtDisplay(topLevel),
            RootWindowOfScreen(XtScreen(topLevel)),
            icon_bits,
            icon_width, icon_height );
    XtVaSetValues(topLevel, XtNiconPixmap, icon_pixmap, NULL);
        .
        .
        .
    /* realize widgets */
}
```

The included file, *icon*, is in standard X11 bitmap file format. You can create such a file using the *bitmap* application from the standard distribution, or *xbitmap8* from the examples for this book.

The window manager may have a preferred standard size for icons. If so, it will set a property. The application can read this property with the Xlib call `XGetIconSizes`. To fully support a variety of window managers, an application should be capable of creating icon pixmaps of different sizes, depending on the values returned by `XGetIconSizes`.

An application has the option of creating its own icon window, and then passing its ID to the window manager for management. This might be done so that the application can draw a multi-colored picture in the icon, instead of the traditional two-color bitmap, or animate the icon. However, as with all hints, the window manager is not guaranteed to honor your desires and may just ignore the icon window you provide. But if you want to try anyway, create the window using Xlib calls, and select input on it so that your application receives `Expose` events when they occur. Then set the ID of the window using the `XtNiconWindow` resource, before realizing the application. Then make sure your event loop redraws the icon when it receives `Expose` events on it. Accomplishing this will require writing your own modified version of `XtAppMainLoop` to handle the icon events.

10.1.7 Window Manager Decorations

Virtually all current window managers decorate windows on the screen.* These decorations typically include a title bar for the window with gizmos for moving and resizing the window. Current decorating window managers include *twm*, *awm*, *mwm*, *olwm*, and *gwm*.

The way these decorations are implemented can have an impact on Toolkit applications. The window manager places the decorations in a window slightly bigger than the application, and then *reparents* the application's top-level window into the decoration window. Reparenting gives the top-level window a new parent instead of the root window. The window manager

*The *uwm* window manager doesn't decorate, but it is defunct as of R4 since it doesn't honor the current ICCCM.

actually creates the frame window as a child of the root window, then reparents the application's top-level window into the frame, and then maps the frame window.

Reparenting affects the application mainly when you try to determine a global position (relative to the root window) from the `XtNx` and `XtNy` resources of the Shell widget. This is usually done to place pop ups. Under a nonreparenting window manager such as *uwm*, these coordinates are indeed relative to the root window, because the parent of the shell is the root window. However, under a window manager that reparents, the coordinates of the application's main Shell widget are relative to the decoration window, not the root window. Therefore, these coordinates cannot be used for placing pop ups within the application. The code shown in Section 3.3 that places a pop up uses `XtTranslateCoords` instead of relying on the position resources.

10.2 Selections: Widget-To-Widget Communication

Selections are a general mechanism for communicating information between two clients of the same server. The most familiar example of selections is selecting text from one *xterm* application and pasting it into another *xterm* application. This text can also be pasted into an *xmh* mail composition window, or into an *xedit* application.

In Xt applications, selection is normally implemented within widgets. For example, the Athena Text widget is used by *xmh* and *xedit*, and it is this widget that supports cutting and pasting of text in these applications. The application code of *xmh* and *xedit* does not play a part in the communication of selection data. The fact that selection is implemented in widgets also means that two widgets within the same application can communicate with each other through selections. For example, the Text widget is sometimes used to provide single-line input fields in an application. Since the Text widget supports selections, the user could select the text in one field and paste it into another field. This feature would be present without any code in the application. If the same widget class supports both copying and pasting, selections can be used to move data in a single widget. For example, selections allow you to copy text within an editor in *xterm* and paste the text in a new place (although some editors require keyboard commands to position the insertion point).

The selection mechanism is not limited to text. It requires only that the sender and recipient have knowledge of the format of the data being transferred. Therefore, selections can be used to transfer graphics between widgets that can understand a common format for communicating graphics. Unfortunately, a standard format for graphics selections has not yet been agreed upon.

The selection mechanism uses properties for transferring data and uses the `SelectionClear`, `SelectionNotify`, and `SelectionRequest` event types to synchronize the communication. Essentially, a property is common storage. Properties are stored in the server, and are named pieces of data associated with a window on that particular server, that any client of that server can read or write. Because the various clients may not be running on the

same host, all communication between applications and between applications and the window manager must take place through the server using properties.*

The basic routines used to implement selections are part of Xlib. Xt provides an interface to the Xlib selection routines that makes selections easier to use in the context of widget code. Since there is a maximum size for the data stored in a single property, communication of large blocks of data using Xlib requires several partial transfers. Xt supplies routines that transparently perform the multiple partial transfers so that they appear to the application program like a single transfer, called an *atomic* transfer.

Xt also supplies a separate set of parallel routines that transfer a large selection one chunk at a time so that it appears as multiple small selections. This is called an *incremental* transfer. Incremental transfer can be preferable on systems with limited memory and more natural when the data is normally stored in small chunks.

The Toolkit selection routines also have built-in timeouts, so that one application won't wait forever for another to provide data.

First we'll give you an overview of how an atomic selection transaction works, and then we'll discuss and demonstrate how to write the code to implement atomic selections in a widget. Following this is a discussion of how incremental selections work, and finally a discussion of the standard selection formats (known as target types).

If you are writing a custom widget that contains data that could be pasted into other instances of the widget or other widgets, you should read on to see how to implement selections. Otherwise, the rest of this chapter is probably only of academic interest to you.

10.2.1 How Atomic Selection Works

Selections communicate between an *owner* widget and a *requestor* widget.† The owner has the data and the requestor wants it. The owner is the widget in which the user has selected something, and the requestor is the widget in which the user has clicked to paste that something. Many widgets need to act as both owner and requester at different times. The code to handle each of the two roles is separate.

Here is a brief overview of the steps that take place during a single transfer of data from one widget to another. We'll assume we have two instances of the same widget class, which can operate either as the owner or the requestor. Initially, both widgets are in exactly the same state, neither having any text selected, and neither being the owner or requestor. We'll also assume that selections are implemented using actions tied to pointer buttons, as in existing widgets that use selections.

---

*It is not practical to write your own networking routines in an application to communicate with other clients running on different hosts because your client may be communicating with the server using TCP/IP, and the other client may be using DECnet. Both may be the same X application that you wrote, but compiled with a version of Xlib that uses a different protocol layer underneath the X protocol. When you communicate through the server using properties, the server takes care of the translation.

†Note that since selections can be implemented in the application, as they are by non-Xt applications, the words "application" and "widget" are interchangeable in this section.

- The user selects an item in Widget *A*, and the widget highlights the item. An action, typically called in response to a <Btn1Down> translation, marks the start of a selection. A subsequent <Btn1Motion> event (that is, a motion event with button 1 held down) invokes another action to extend the highlighted selection.
- A <Btn1Up> event invokes a third action in Widget *A* that actually makes the selection. The action calls XtOwnSelection to claim ownership of the PRIMARY selection for Widget *A*. This means that Widget *A* claims the sole right to use the XA_PRIMARY property for communication with other widgets, until some other widget claims ownership. (More about XA_PRIMARY below.) The call to XtOwnSelection also registers three procedures, to be called by Xt in response to selection events. The *lose_ownership_proc* handles the case where Widget *A* loses the selection (because the user has made another selection elsewhere), the *convert_proc* converts the data in the selection to the target property type requested by Widget *B* (see below), and the optional *transfer_done_proc* prepares for the next selection request, if necessary (this function is often NULL).
- The user pastes the item into Widget *B*, usually by clicking a pointer button. (By convention, a translation for <Btn2Down> invokes the action to do this.)
- The paste action in Widget *B* requests the value of the current selection by calling XtGetSelectionValue; this specifies a *target* type, and a *requestor_callback* that will be invoked to actually paste the data when Widget *A* reports that the conversion has been successfully completed.
- The XtGetSelectionValue call by Widget *B* also generates a SelectionRequest event. In response to this event, Xt invokes the *convert_proc* registered by the call to XtOwnSelection in Widget *A*.
- The *convert_proc* in Widget *A* converts the selected item into the appropriate data type for the target property specified by *B*, if possible. The converted data is stored in the XA_PRIMARY property.
- Based on the return values from Widget *A*'s conversion procedure, Xt sends a SelectionNotify event to inform Widget *B* whether or not the data was successfully converted.
- The *requestor_callback* registered in Widget *B* reads the data from the property and displays it, or if Widget *A* reported that the conversion could not be made, Widget *B* beeps or otherwise indicates that the kind of data selected in Widget *A* cannot be pasted in Widget *B*, or that the kind of data requested by *B* cannot be supplied by *A*.
- Xt notifies Widget *A* that the selection has been transferred, so that the widget's *transfer_done_proc* can disown the selection if the selection is of a kind that can be transferred only once.

Figure 10-2 shows this procedure in graphic form.

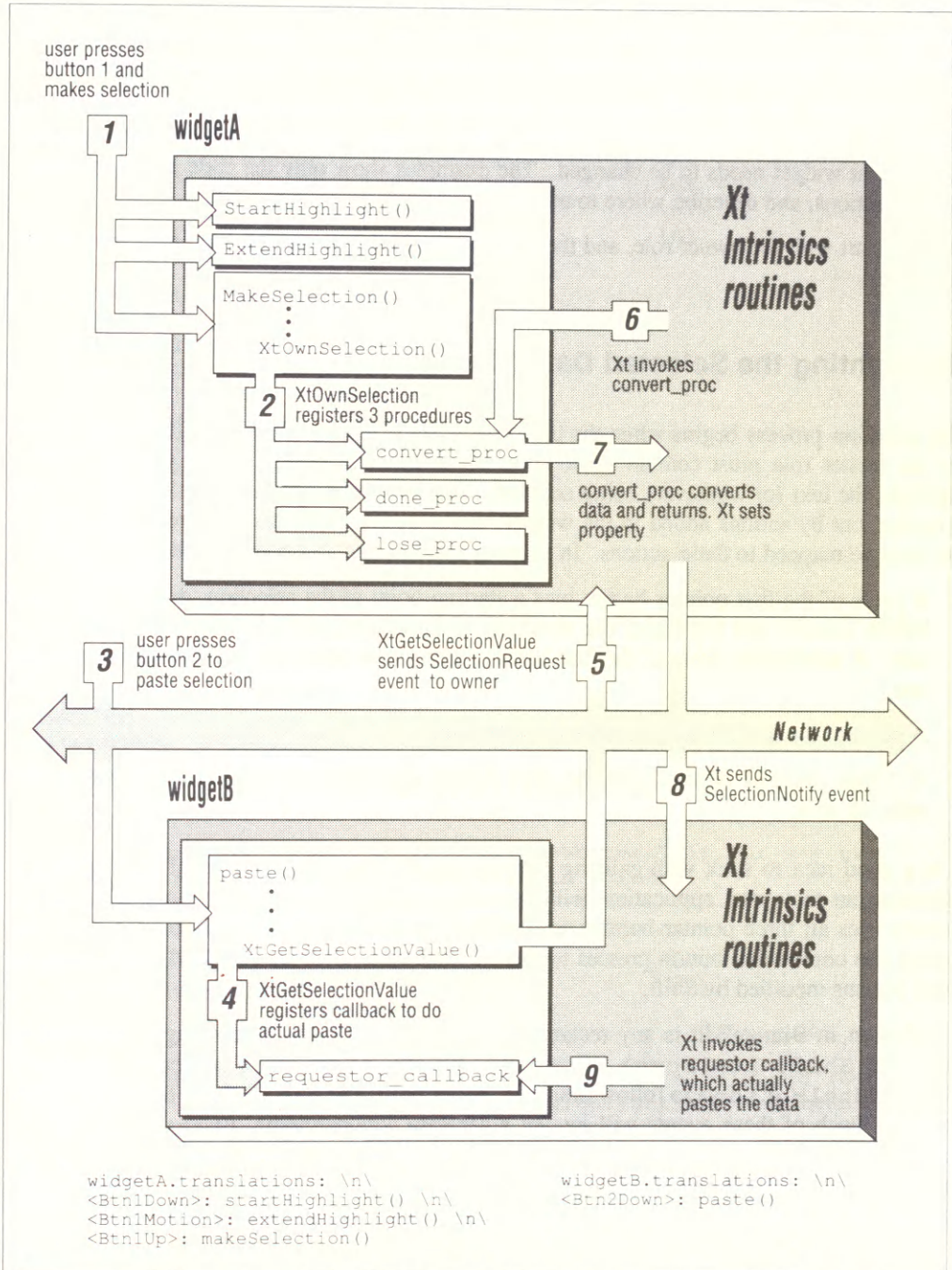


Figure 10-2. The process of selection transfer

The code to implement selections is divided logically into a number of functions within the widget. The next sections show the code necessary to implement both the owner and the requestor roles. The code for each role is separate and cannot share any widget variables with the other role because the transaction may be between two different widget instances.

As an example, we will add support for selections to the BitmapEdit widget described in Chapter 5, *Inside a Widget*, and Chapter 6, *Basic Widget Methods*. Little of the existing code in that widget needs to be changed. The examples show only the code added to implement selections, and describe where to add it.

We will start with the owner role, and then proceed to the requestor role, and then back to the owner role.

10.2.2 Highlighting the Selected Data (Owner)

The selection process begins when the user selects some text in Widget *A*. The widget code for the owner role must contain some code to identify the area or items selected and to highlight the text (or whatever) being selected. This user-interface portion of the owner role is carried out by actions added to the widget. Some event sequences that are not already in use must be mapped to these actions. In existing applications that support selections of text:

- A press of the first pointer button pins a starting point of the selection, dragging the first button extends and highlights the selection, and releasing the first button ends the selection. A subsequent press of the first button starts a new selection, unhighlighting the old one.
- A press of the second button pastes the selection.
- A button press or motion with the third button depressed repositions either end of the selected area.

It is a good idea to stick with existing conventions for the user interface of highlighting a selection, so that your application will operate in a familiar way. But since BitmapEdit already uses all three pointer buttons, unmodified, to set, unset, and toggle bitmap cells, we cannot use unmodified button presses to control selections. But we can (and will) use these same buttons modified by Shift.

A selection in BitmapEdit is any rectangular set of bitmap cells. Accordingly, BitmapEdit will use `Shift<Btn1Down>` to set the top-left corner of a selection rectangle, `Shift<Btn1Motion>` to follow dragging, and `Shift<Btn1Up>` to set the bottom right corner.* Each of these events will invoke a separate action routine, `StartHighlight`, `ExtendHighlight`, and `MakeSelection`, respectively. These translations are added to the widget's default translation table, and the actions are added to the actions table. Example 10-3 shows the action table, the translation table, and the three action routines.

*For simplicity, we won't copy the third button semantics used in text selections. This could easily be added.

Example 10-3. BitmapEdit: actions that highlight selection

```
static char defaultTranslations[] =
        "Shift<Btn1Down>:   StartHighlight()    \n\
        Shift<Btn1Motion>:  ExtendHighlight()   \n\
        Shift<Btn1Up>:      MakeSelection()     \n\
        Shift<Btn2Down>:    PasteSelection()    \n\
        ~Shift<Btn1Down>:   DoCell()            \n\
        ~Shift<Btn2Down>:   UndoCell()          \n\
        ~Shift<Btn3Down>:   ToggleCell()        \n\
        ~Shift<Btn1Motion>: DoCell()            \n\
        ~Shift<Btn2Motion>: UndoCell()          \n\
        ~Shift<Btn3Motion>: ToggleCell()";

static XtActionsRec actions[] = {
    {"DoCell", DoCell},
    {"UndoCell", UndoCell},
    {"ToggleCell", ToggleCell},
    {"StartHighlight", StartHighlight},
    {"ExtendHighlight", ExtendHighlight},
    {"MakeSelection", MakeSelection},
    {"PasteSelection", PasteSelection},
};
  .
  .
  .
/*
 *  User presses first button (by default), starting highlighting.
 */
static void
StartHighlight(w, event)
Widget w;
XButtonEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    cw->bitmapEdit.first_box = False;

    cw->bitmapEdit.select_start_x = (cw->bitmapEdit.cur_x + event->x)
            / cw->bitmapEdit.cell_size_in_pixels;
    cw->bitmapEdit.select_start_y = (cw->bitmapEdit.cur_y + event->y)
            / cw->bitmapEdit.cell_size_in_pixels;

    /* clear old selection */
    Redisplay(cw, NULL);
}

/*
 * MakeSelection is call when the first button is released (by
 * default).  This finishes the user's highlighting, and means
 * triggers ownership of the PRIMARY selection.
 */
static void
MakeSelection(w, event)
Widget w;
XButtonEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int temp;

    cw->bitmapEdit.select_end_x = (cw->bitmapEdit.cur_x + event->x)
```

Example 10-3. BitmapEdit: actions that highlight selection (continued)

```
            / cw->bitmapEdit.cell_size_in_pixels;
    cw->bitmapEdit.select_end_y = (cw->bitmapEdit.cur_y + event->y)
            / cw->bitmapEdit.cell_size_in_pixels;

    if ((cw->bitmapEdit.select_end_x == cw->bitmapEdit.select_start_x)
            && (cw->bitmapEdit.select_end_y ==
            cw->bitmapEdit.select_start_y))  {
        Redisplay(cw, NULL);
        return; /* no selection */
    }

    /* swap start and end if end is greater than start */
    if (cw->bitmapEdit.select_end_x < cw->bitmapEdit.select_start_x) {
        temp = cw->bitmapEdit.select_end_x;
        cw->bitmapEdit.select_end_x = cw->bitmapEdit.select_start_x;
        cw->bitmapEdit.select_start_x = temp;
    }

    if (cw->bitmapEdit.select_end_y
            < cw->bitmapEdit.select_start_y) {
        temp = cw->bitmapEdit.select_end_y;
        cw->bitmapEdit.select_end_y =
                cw->bitmapEdit.select_start_y;
        cw->bitmapEdit.select_start_y = temp;
    }

    if (XtOwnSelection(cw, XA_PRIMARY, event->time, convert_proc,
            lose_ownership_proc, transfer_done_proc) ==
            False) {
        XtWarning("bitmapEdit: failed attempting to become
                selection owner; make a new selection.\n");
        /* Clear old selection, because lose_ownership_proc
         * isn't registered. */
        Redisplay(cw, NULL);
    }
}

/*
 * ExtendHighlight is called when the mouse is being dragged with
 * the first button down (by default).  During this time, the
 * bitmap cells that are selected (by crosses) are dynamically
 * changed as the mouse moves.
 */
static void
ExtendHighlight(w, event)
Widget w;
XMotionEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    static int last_drawn_x, last_drawn_y;
    int event_cell_x, event_cell_y;

    event_cell_x = cw->bitmapEdit.cur_x + (event->x /
            cw->bitmapEdit.cell_size_in_pixels);
    event_cell_y = cw->bitmapEdit.cur_y + (event->y /
            cw->bitmapEdit.cell_size_in_pixels);

    if ((event_cell_x == last_drawn_x) && (event_cell_y ==
            last_drawn_y))
```

Example 10-3. BitmapEdit: actions that highlight selection (continued)

```
        return;

    if (cw->bitmapEdit.first_box) {
        DrawBoxOfXs(cw, last_drawn_x, last_drawn_y, False);
        DrawBoxOfXs(cw, event_cell_x, event_cell_y, True);
    }
    else {
        DrawBoxOfXs(cw, event_cell_x, event_cell_y, True);
        cw->bitmapEdit.first_box = True;
    }

    last_drawn_x = event_cell_x;
    last_drawn_y = event_cell_y;
}

/*
 *  DrawBoxOfXs fills a rectangular set of bitmap cells with
 *  crosses, or erases them.
 */
static void
DrawBoxOfXs(w, x, y, draw)
Widget w;
Position x, y;
Bool draw;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    Position start_pos_x, start_pos_y;
    Dimension width, height;
    GC gc;
    int i, j;

    start_pos_x = cw->bitmapEdit.cur_x + cw->bitmapEdit.select_start_x;
    start_pos_y = cw->bitmapEdit.cur_x + cw->bitmapEdit.select_start_y;

    /* swap start and end if end is greater than start */
    if (x < start_pos_x) {
        width = start_pos_x - x;
        start_pos_x = x;
    }
    else {
        width = x - start_pos_x;
    }

    if (y < start_pos_y) {
        height = start_pos_y - y;
        start_pos_y = y;
    }
    else {
        height = y - start_pos_y;
    }

    for (i=start_pos_x;i < start_pos_x + width;i++)
         for (j=start_pos_y;j < start_pos_y + height;j++)
              DrawX(cw, i, j, draw);
}

/*
 * DrawX draws an X in a bitmap cell, in white if a black cell, and
 * in black if a white cell.
```

Example 10-3. BitmapEdit: actions that highlight selection (continued)

```
 */
DrawX(cw, x, y, draw)
BitmapEditWidget cw;
Position x, y;
Bool draw;
{
    GC gc;
    if (cw->bitmapEdit.cell[x + y * cw->bitmapEdit.pixmap_width_in_cells]
            == DRAWN)
        if (draw)
            gc = cw->bitmapEdit.deep_undraw_gc;
        else
            gc = cw->bitmapEdit.deep_draw_gc;
    else
        if (draw)
            gc = cw->bitmapEdit.deep_draw_gc;
        else
            gc = cw->bitmapEdit.deep_undraw_gc;

    XDrawLine(XtDisplay(cw), XtWindow(cw), gc,
            x * cw->bitmapEdit.cell_size_in_pixels,
            y * cw->bitmapEdit.cell_size_in_pixels,
            (x + 1) * cw->bitmapEdit.cell_size_in_pixels,
            (y + 1) * cw->bitmapEdit.cell_size_in_pixels);

    XDrawLine(XtDisplay(cw), XtWindow(cw), gc,
            x * cw->bitmapEdit.cell_size_in_pixels,
            (y + 1) * cw->bitmapEdit.cell_size_in_pixels,
            (x + 1) * cw->bitmapEdit.cell_size_in_pixels,
            y * cw->bitmapEdit.cell_size_in_pixels);
}
```

The actions shown in Example 10-3 reference some new instance part fields we have added to *BitmapEditP.h* to store the state of the selection. Four of these variables are the *x* and *y* coordinates in cells of the top-left and bottom-right corners of the highlighted area: `select_start_x`, `select_start_y`, `select_end_x`, and `select_end_y`. These coordinates will be used throughout the owner code. (Because these fields are in units of bitmap cells, not pixels, their type is `int` rather than `Position`.)

The `StartHighlight` action is quite simple. It sets the upper-left corner instance part fields based on the button press position in the triggering event. It also clears the highlighting of the old selection, if one exists, and sets a flag to indicate that this is the beginning of a new selection. The `first_box` flag is added as an instance part field because it will be needed in the `ExtendHighlight` action.

`ExtendHighlight` is responsible for drawing Xs dynamically on the selected cells, much as the window manager draws the outline of a window while it is being moved. Xs are drawn on each cell in a rectangle of bitmap cells beginning from the cell selected in the `Start-Highlight` action and ending at the cell the pointer button was released in. `Extend-Highlight` is triggered by pointer motion events that occur while the first button is held down. It calculates which bitmap cell the pointer is in, and if the cell is not the same as the cell the pointer was in the last time `ExtendHighlight` was called, it erases the previous Xs and draws new ones. This is done using two different GCs—one for drawing in black and the other in white, so the color that contrasts with the current state of each cell can be used.

The black GC is also used to copy the main pixmap to the screen, and the latter is just for doing the highlighting.

Code was added to the `initialize` method to create this latter GC. (See the example code distribution for this modified widget instance structure.)

The `MakeSelection` action is triggered when the button is released after dragging, indicating that the selection is complete. This action sets the `select_end_x` and `select_end_y` instance part variables to reflect the bottom-right corner of the selection, and then swaps the top-left and bottom-right coordinates if the end coordinate is to the left or above the start coordinate. If the start and end coordinates are the same, then the action returns, since no selection was made. If a selection was made, `MakeSelection` calls `XtOwnSelection`.

10.2.3 Making the Selection with XtOwnSelection (Owner)

Once the area is highlighted, Widget *A* calls `XtOwnSelection` to assert that it wants the right to set the value of a property that will be used to transfer information.

`XtOwnSelection` is called with six arguments:

```
Boolean XtOwnSelection(widget, selection, time, convert_proc,
        lose_ownership_proc, transfer_done_proc)
    Widget widget;
    Atom selection;
    Time time;
    XtConvertSelectionProc convert_proc;
    XtLoseSelectionProc lose_ownership_proc;
    XtSelectionDoneProc transfer_done_proc;
```

The *`selection`* argument specifies an *Atom*—a number representing a property. Properties are arbitrarily named pieces of data stored on the server. To simplify communication with the server, a property is never referenced by name, but by a unique integer ID called an atom. Standard atoms are defined in *<Xatom.h>* using defined symbols beginning with `XA_`; nonstandard atoms can be obtained from the server by calling the Xlib function `XtInternAtom`.

Widgets that support one selection at a time pass the predefined `XA_PRIMARY` atom to `XtOwnSelection`. The ICCCM also allows you to use the `XA_SECONDARY` and `XA_CLIPBOARD` atoms. `XA_SECONDARY` would be used by widgets implementing behavior involving more than one selection (for example, allowing the user to make two selections, and to exchange the data they contain). No current clients implement this behavior. The `XA_CLIPBOARD` atom should be used by a widget that allows the user to delete a selection. We'll talk more about the properties referenced by the `XA_SECONDARY` and `XA_CLIPBOARD` atoms in Section 10.2.9.3.

The purpose of the `XtOwnSelection` call is to make sure that only one widget has the right to set the `XA_PRIMARY` selection property at a time, and to assert that this widget is prepared to honor requests for this data. Notice that when you select text with *xterm*, that text is highlighted only until you select a different area in a different window.

The next three arguments to XtOwnSelection specify procedures that will carry out essential parts of the selection operation:

- The *convert_proc* is responsible for converting the selected data into the representation requested by the requestor in its call to XtGetSelectionValue; this function is called by Xt when a SelectionRequest event arrives, as a result of the requestor calling XtGetSelectionValue. The *convert_proc* is of type XtConvertSelectionProc.
- The *lose_ownership_proc* clears the highlighted area when this widget has lost the selection (because some other widget has taken it); this routine is called by Xt when a SelectionClear event arrives. The *lose_ownership_proc* is of type XtLoseSelectionProc.
- The *transfer_done_proc* is called by Xt when the requestor has successfully retrieved the converted value. If this selection is intended to be erased after being transferred once, this function should free any storage allocated in the transfer. If, instead, the owner remains ready for pasting the same selection into other widgets, this function pointer should be NULL. (For example, *xterm* does not clear its selection after pasting once.) The *transfer_done_proc* is of type XtSelectionDoneProc.

The XtOwnSelection call also takes a *time* argument—this should be the time from the event that completed the highlighting, not the constant CurrentTime. If XtOwnSelection returns FALSE, it means that because of network conditions another client has called XtOwnSelection with a more recent time argument.

XtOwnSelection returns TRUE or FALSE to indicate whether Widget *A* has been granted ownership. If TRUE, and if this widget was not already the owner, then the old owner (if any) will receive a SelectionClear event and will clear its highlighted area. The process may end here if the user never pastes the selected data anywhere. If the user selects a different piece of data, the selection owner will receive a SelectionClear itself before ever having converted the data for a requestor.

Otherwise the owner code waits to hear from the requestor that it is ready to paste the selection.

10.2.4 Requesting the Selection (Requestor)

When the user pastes the selection into Widget *B*, Widget *B* becomes the requestor and the second part of the process begins. First of all, Widget *B* needs a translation that maps a certain key or button event to an action that pastes data. This action calls XtGetSelectionValue.

XtGetSelectionValue is called with six arguments:

```
void XtGetSelectionValue (widget, selection, target, callback,
    client_data, time)
Widget widget;
Atom selection;
Atom target;
XtSelectionCallbackProc callback;
```

```
XtPointer client_data;
Time time;
```

The *selection* argument is an atom specifying which property is being used to transfer the selection. This will typically be XA_PRIMARY, though in theory two widgets (or two instances of the same widget) could agree to transfer some particular data using other properties.

The *target* argument is another atom, this one specifying a target representation type in which the requestor wants the information. We'll talk more about the possible values for this atom in a moment.

The *callback* argument is a pointer to a callback procedure that actually pastes the data. Xt will call this procedure when the owner has converted the data. The XtGetSelectionValue call registers the callback with Xt, and sends a SelectionRequest event to the owner.

When the selection owner has successfully converted the data, the owner sends back a SelectionNotify event to Xt. Xt then calls the requestor's callback procedure. The callback procedure must handle the pasting of the data into the widget's data structures, and it must handle the case where the data could not be converted into the requested representation. We'll discuss the responsibilities of this procedure in more detail once we've seen the owner's conversion procedure.

The *client_data* argument of XGetSelectionValue is normally used to pass the event that triggered the pasting into the requestor callback. The callback will use the coordinates at which the event occurred as the location at which to paste the data.

As in the call to XtOwnSelection, the *time* argument should be the time from the event that initiated the pasting, not the constant CurrentTime.

10.2.4.1 Possible Target Type Atoms

The *target* argument to XtGetSelectionValue is an atom that the requestor uses to tell the owner what kind of information it is looking for from the selection.* This is not necessarily a conversion of the actual selection data. For example, it might be a timestamp on the data, or some characteristic of it, such as its size or font.

To take full advantage of the Xt selection mechanism, you need to understand what atoms can be used as selection targets. However, apart from some standard, predefined atoms, the atom for a property is not known by a client until it queries the server for the atom using an XInternAtom call. This call specifies the string name of the property and returns the atom.

Some atoms needed by almost all applications are predefined in the header file *<X11/Xatom.h>*. These atoms are symbolic constants that can be used without an XInternAtom call. The constants always start with the prefix XA_. See *<X11/Xatom.h>* for the complete list of predefined atoms.

*As described earlier, atoms (rather than strings) are always used in network transmissions to identify properties.

Note that many of the predefined atoms have uses in X other than as target types, and not all are appropriate as selection targets. A few of those that might be useful as targets include:

```
XA_STRING
XA_INTEGER
XA_BITMAP
```

At present, the Athena widgets and the clients in the MIT core X distribution use only XA_STRING as a target. However, one can imagine uses for other targets as well. For example, XA_FONT might be used as a target to indicate that the requestor doesn't want the text of a selection, but wants only to know its font. XA_TIMESTAMP might be used to indicate that the requestor wants a timestamp for the data. Once the XtGetSelectionValue request is made, a SelectionRequest event is generated, which Xt uses to invoke the owner widget's *convert_proc*. The *convert_proc* will need to branch according to the target type passed in by the requestor through Xt, and convert the data accordingly before transferring the selection to the requestor.

For selection of multiple data types to work correctly between any two arbitrarily chosen widgets, there must be conventions about which targets will be supported. As a step in the direction of interwidget selection compatibility, the ICCCM specifies a required target type of XA_TARGETS, to which the owner is supposed to respond by returning a list of the target types into which it is capable of converting data. We'll talk about how to work with this target in Section 10.2.9.

In a custom widget such as BitmapEdit, which has a custom data type not represented by a predefined atom, it is possible to obtain a custom atom using XInternAtom, and use that as the target. Because the XInternAtom call requires a round trip to the server, the best place to do this is in the widget's initialize method.* The returned atom can be stored in a field added to the widget's instance part. BitmapEdit's target type, "CELL_ARRAY," is unique to BitmapEdit. The atom is stored in an instance part field called target so that it is available in *convert_proc* and in the requestor role code. Example 10-4 shows the code added to the initialize method to get an atom for the string "CELL_ARRAY."

Example 10-4. BitmapEdit: getting the atom for a widget-specific target type

```
/* ARGSUSED */
static void
Initialize(request, new)
BitmapEditWidget request, new;
{
        .
        .
        .
    new->bitmapEdit.target_atom = XInternAtom(XtDisplay(new),
            "CELL_ARRAY," FALSE);
```

The target type property name used by BitmapEdit is "CELL_ARRAY."

*Since Xlib functions causing round-trip requests can impact performance, they should be called only once if possible, and where possible, during the set-up phase of the application as opposed to the event-loop phase. If XInternAtom were called every time the user made a selection, it would slow the application. In addition, since atoms are server resources that are not freed until the server shuts down, it is important to use predefined atoms whenever possible.

The third argument to XInternAtom is a Boolean that indicates what to do if the atom doesn't already exist. If this argument is TRUE, XInternAtom will return None if no other client has already initialized this atom. This argument should always be FALSE since your widget might be the first one to try to make a selection using this target type atom.

Note that for repeated calls to XtInternAtom with the same string as an argument, even from different widgets, the returned atom will be the same. There will be only one atom created for any unique string interned on a given server. (Case is important: "Cell_Array" would return a different atom than "CELL_ARRAY".)

This XInternAtom call occurs in every instance of the BitmapEdit widget, since all instances need to know the atom to participate in selections using that target type. Both the owner and requestor widgets, in separate applications, will get the same atom in return.

10.2.4.2 The Paste Action from BitmapEdit

To initiate the requestor role, you need to assign an event sequence to trigger the pasting, write an action that calls XtGetSelectionValue, and write a callback function that inserts the returned data.

Traditionally, the action to paste data is triggered by a press of the second pointer button. BitmapEdit will use the shifted second button since it uses the unshifted button for other purposes. Example 10-5 shows the action mapped to Shift<Btn2Down>.

Example 10-5. BitmapEdit: action to paste a selection

```
static void
PasteSelection(w, event)
Widget w;
XButtonEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XtGetSelectionValue(cw, XA_PRIMARY, cw->bitmapEdit.target_atom,
            requestor_callback, event,
            event->time);
}
```

This action can be dropped verbatim into your widget, replacing only the widget name and the name of the instance fields you are using to store the target atom.

10.2.5 Converting the Selection (Owner)

The real challenge in handling selections is writing the *convert_proc*, which converts data to the format specified by the requestor and prepares it for transfer.

As mentioned above, the widget's *convert_proc* is called by Xt when the requestor calls XtGetSelectionValue. The procedure is passed the selection atom and the target atom, and is expected to return in its arguments the type, value, size, and format of the converted selection data.

To support the possibility of having to convert the data to any one of several targets, the code branches according to the value of the target passed in, and does its conversions accordingly. In the version of BitmapEdit's *convert_proc* shown here, only one target type is handled. Section 10.2.9.2 describes how to add more target types, including some that are required by the ICCCM such as TARGETS.

Once the conversion is made, the procedure sets the *value_return* argument passed in to a pointer to a block of memory, and *length_return* to the size of the block. This block of memory will be set into a property by Xt and passed to the requestor's callback in the same form—as a pointer to a block of memory and a length. This puts constraints on the formats that can be used for the data.

For text selections, the data is usually a simple character string. The *convert_proc* simply needs to set *value_return to a pointer to the string, and length_return to the length of the string. For BitmapEdit, however, the required data is a string (of bitmap cell states) *plus* width and height values. Since C provides easy ways to put numeric values into strings and to get them out again at the other end, we've chosen to handle this data by converting the numbers to characters and tacking them on to the beginning of the string.

If your selection is composed of a number of numeric values, you can create a structure containing the values and then pass a pointer to the structure. However, the structure cannot contain pointers, because the data pointed to by these pointers will not be set into the selection property. For example, BitmapEdit cannot pass a pointer to a structure containing a string pointer field and width and height fields because the block of memory pointed to by the string pointer will not be copied.

In this case, the *type_return* can simply be the target atom that was passed in, indicating that the data is of the requested type.

The *format_return* is the size in bits of each element in the array. Since we are passing a compound string, this value is 8. If we were passing a structure, *length_return* would be 1 and *format_return would be 8 times the size of the structure in bytes.

Example 10-6 shows BitmapEdit's *convert_proc*.

Example 10-6. BitmapEdit: converting the selection value

```
static Boolean
convert_proc(w, selection, target, type_return, value_return,
        length_return, format_return)
Widget w;
Atom *selection;
Atom *target;
Atom *type_return;
XtPointer *value_return;
unsigned long *length_return;
int *format_return;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int x, y;
    int width, height;

    /* handle all required atoms, and the one that we use */
    /* Xt already handles MULTIPLE, no branch necessary */
    if (*target == XA_TARGETS(XtDisplay(cw))) {
```

Example 10-6. BitmapEdit: converting the selection value (continued)

```
        /* Required atom: see Example 10-10 */
          .
          .
          .
    }
    else if (*target == cw->bitmapEdit.target_atom) {
        char *data;

        width = cw->bitmapEdit.select_end_x - cw->bitmapEdit.select_start_x;
        height = cw->bitmapEdit.select_end_y - cw->bitmapEdit.select_start_y;

        /* 8 chars is enough for two 3-digit numbers and two delimiters */
        *length_return = ((width * height) + 8) * sizeof(char);

        data = XtMalloc(*length_return);

        sprintf(data, "%d@%d~", width, height);

        for (x = 0; x < width; x++) {
            for (y = 0; y < height; y++) {
                data[8 + x + (y * width)] = cw->bitmapEdit.cell[(x +
                        cw->bitmapEdit.select_start_x) +
                        ((y + cw->bitmapEdit.select_start_y) *
                        cw->bitmapEdit.pixmap_width_in_cells)];
            }
        }

        *value_return = data;

        *type_return = cw->bitmapEdit.target_atom;

        *format_return = 8;  /* number of bits in char */
        return(True);
    }
    else {
        /* code to handle standard selections: see Example 10-10 */
          .
          .
          .
    }
}
```

This code determines the width and height of the selected rectangle from the instance part fields, and then allocates enough memory to fit a character array big enough to fit the width and height values, delimiters, and a width by height character array. Then it copies the current contents of the selected area into the allocated character array. Finally, it sets `*value_return` to point to the compound string, and sets `*length_return` to the length of this string. `*format_return` is the size in bits of each element in the array, which in this case is 8 bits.

10.2.6 Finally Pasting the Selection (Requestor)

When the owner's *convert_proc* returns, Xt sends a SelectionNotify event to the requestor. The Xt code on the requestor side then invokes the callback routine the requestor registered with the call to XtGetSelectionValue.

The *requestor_callback* function is passed all the same arguments that the owner received in the *convert_proc*, plus the values that the owner returned through the argument of *convert_proc*. In the BitmapEdit widget, it sets instance part fields to paste the data.

Example 10-7 shows the requestor callback function from BitmapEdit.

Example 10-7. BitmapEdit: pasting selection in requestor_callback function

```
/* ARGSUSED */
static void
requestor_callback(w,client_data,selection,type,value,length,format)
Widget w;
XtPointer client_data;  /* cast to XButtonEvent below */
Atom *selection;
Atom *type;
XtPointer value;
unsigned long *length;
int *format;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    if ((*value == NULL) && (*length == 0)) {
        XBell(XtDisplay(cw), 100);
        XtWarning("bitmapEdit: no selection or selection timed out:\
              try again\n");
    }
    else {
        XButtonEvent *event = (XButtonEvent *) client_data;
        int width, height;
        int x, y;
        int dst_offset_x, dst_offset_y;
        char *ptr;

        dst_offset_x = (cw->bitmapEdit.cur_x + event->x) /
                cw->bitmapEdit.cell_size_in_pixels;
        dst_offset_y = (cw->bitmapEdit.cur_y + event->y) /
                cw->bitmapEdit.cell_size_in_pixels;

        printf("dst offset is %d, %d\n", dst_offset_x, dst_offset_y);

        ptr = (char *) value;
        width = atoi(ptr);
        ptr = index(ptr, '@');
        ptr++;
        height = atoi(ptr);
        ptr = &value[8];

        for (x = 0; x < width; x++) {
            for (y = 0; y < height; y++) {
                /* range checking */
                if (((dst_offset_x + x) >
```

Example 10-7. BitmapEdit: pasting selection in requestor_callback function (continued)

```
                        cw->bitmapEdit.pixmap_width_in_cells)
                        || ((dst_offset_x + x) < 0))
                    break;
                if (((dst_offset_y + y) >
                        cw->bitmapEdit.pixmap_height_in_cells)
                        || ((dst_offset_y + y) < 0))
                    break;
                cw->bitmapEdit.cell[(dst_offset_x + x)
                        + ((dst_offset_y + y) *
                        cw->bitmapEdit.pixmap_width_in_cells)]
                        = ptr[x + (y * width)];
                if (cw->bitmapEdit.cell[(dst_offset_x + x)
                        + ((dst_offset_y + y) *
                        cw->bitmapEdit.pixmap_width_in_cells)]
                        == DRAWN)
                    DrawCell(cw, dst_offset_x + x,
                            dst_offset_y + y,
                            cw->bitmapEdit.draw_gc);
                else
                    DrawCell(cw, dst_offset_x + x,
                            dst_offset_y + y,
                            cw->bitmapEdit.undraw_gc);
            }
        }
        /* Regardless of the presence of a
         * transfer_done_proc in the owner,
         * the requestor must free the data passed by
         * Xt after using it. */
        XtFree(value);
        Redisplay(cw, NULL);
    }
}
```

The *requestor_callback* first determines whether the conversion was a success by checking the value and data length. If it was not, it beeps and prints a message. This can happen if no selection has been made, or if there is some delay that causes the selection to have timed out before the owner could convert the data.

If the conversion was a success, the *requestor_callback* pastes the data. In BitmapEdit's case, the requestor must first convert the data into a more useful form. This means extracting the width and height values, and then setting the widget's cell array based on the data in the character array passed in. Note that the code should check to make sure that the data can be pasted in the desired position. BitmapEdit must make sure that no attempt is made to set a cell array member outside of the bitmap. The routine then updates the screen display of the widget.

The final responsibility of the *requestor_callback* is to free the memory passed in by Xt.

10.2.7 If the Selection is Lost (Owner)

If the owner loses the selection, either because the selection timed out or because the user made a different selection, the *lose_ownership_proc* that was registered with its call to XtOwnSelection will be invoked. Typically, this function simply clears any highlighting or other visual feedback about the selection, and resets to their initial state any internal variables used in handling selections.

Example 10-8 shows the *lose_ownership_proc* from the BitmapEdit widget.

Example 10-8. BitmapEdit: the lose_ownership_proc

```
/* ARGSUSED */
static void
lose_ownership_proc(w, selection)
Widget w;
Atom *selection;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;

    /* clear old selection */
    cw->bitmapEdit.first_box = FALSE;
    cw->bitmapEdit.select_start_x = 0;
    cw->bitmapEdit.select_start_y = 0;
    cw->bitmapEdit.select_end_x = 0;
    cw->bitmapEdit.select_end_y = 0;
    Redisplay(cw, NULL);
}
```

10.2.8 When the Selection Transfer is Complete (Owner)

The *transfer_done_proc*, registered in the call to XtOwnSelection, is called when the transfer is complete. Its job is to do any processing necessary to get ready for the next selection request. In many cases no processing is necessary and this function can be NULL in the call to XtOwnSelection. However, this function might clear variables or free memory. Some transfers are intended to be made only once to make sure that there is no duplication of information. In these cases, the *transfer_done_proc* would do a lot more, including erasing the visual selection, calling XtDisownSelection (and perhaps even erasing the data that was selected).

10.2.9 ICCCM Compliance

For any two widgets to be able to transparently transfer different types of data, there must be agreement about the possible target types and their contents. The ICCCM suggests a list of possible target types, as shown in Table 10-2. If you can, use one of these target types since this will increase the chances that your widget will be able to communicate with widgets written by other people.

Table 10-2. Target Types Suggested in ICCCM

Atom	Type	Meaning
TARGETS	ATOM	List of valid target atoms
MULTIPLE	ATOM_PAIR	Look in the ConvertSelection property
TIMESTAMP	INTEGER	Timestamp used to acquire selection
STRING	STRING	ISO Latin 1 (+TAB+NEWLINE) text
TEXT	TEXT	Text in owner's encoding
LIST_LENGTH	INTEGER	Number of disjoint parts of selection
PIXMAP	DRAWABLE	Pixmap ID
DRAWABLE	DRAWABLE	Drawable ID
BITMAP	BITMAP	Bitmap ID
FOREGROUND	PIXEL	Pixel value
BACKGROUND	PIXEL	Pixel value
COLORMAP	COLORMAP	Colormap ID
ODIF	TEXT	ISO Office Document Interchange Format
OWNER_OS	TEXT	Operating system of owner
FILE_NAME	TEXT	Full path name of a file
HOST_NAME	TEXT	See WM_CLIENT_MACHINE
CHARACTER_POSITION	SPAN	Start and end of selection in bytes
LINE_NUMBER	SPAN	Start and end line numbers
COLUMN_NUMBER	SPAN	Start and end column numbers
LENGTH	INTEGER	Number of bytes in selection
USER	TEXT	Name of user running owner
PROCEDURE	TEXT	Name of selected procedure
MODULE	TEXT	Name of selected module
PROCESS	INTEGER, TEXT	Process ID of owner
TASK	INTEGER, TEXT	Task ID of owner
CLASS	TEXT	Class of owner—see WM_CLASS
NAME	TEXT	Name of owner—see WM_NAME
CLIENT_WINDOW	WINDOW	Top-level window of owner
DELETE	NULL	TRUE if owner deleted selection
INSERT_SELECTION	NULL	Insert specified selection
INSERT_PROPERTY	NULL	Insert specified property

Because not every widget will support every possible target type, the ICCCM specifies a target type of `XA_TARGETS`, to which the owner is required to respond by returning a list of the target types into which it is capable of converting data.

Normally, a requestor would first call `XtGetSelectionValue` for `XA_TARGETS`, and then in the callback determine which target it wants to request from the list, and then call `XtGetSelectionValue` again for the desired target with a separate callback to process the actual data. This is really two separate selection transfers.

XA_TARGETS is not a predefined atom.* To use it, you must use the Xmu atom caching mechanism described in Section 10.2.9.1.

Fortunately, there is existing template code that you can copy to handle XA_TARGETS and some other standard target types that are required by the ICCCM.

This template code uses the Xmu routine XmuConvertStandardSelection. It also uses an Xmu atom-caching facility that eliminates the need for you to make XInternAtom calls for each of the ICCCM standard target atoms.

10.2.9.1 Xmu Atom Caching

Xmu's caching facility uses symbols similar to those defined in *<X11/Xatom.h>*, except that in this case they are macros that take an argument and call XmuInternAtom. For example, the macro for XA_TARGETS is defined as follows in *<X11/Xmu.h>*:

```
#define XA_TARGETS(d)          XmuInternAtom(d, _XA_TARGETS)
```

where d refers to a pointer to a Display structure. (This can be returned by the XtDisplay macro.) The XmuInternAtom function first tries to get the atom for the string "XA_TARGETS" from Xmu's internal cache. If Xmu doesn't yet have a value for the atom, it calls XInternAtom to make a server request. Because this facility makes only one query to the server, you can access the atoms in this way every time a selection is made without significant penalty. This allows you to place the XA_TARGETS() macro in your selection code instead of adding an instance part variable and setting it in the initialize method.

You might use this macro as follows in a *convert_proc* branch dedicated to handling the TARGETS target type:

```
if (*target == XA_TARGETS(XtDisplay(w))) { ...
```

The Xmu atom caching mechanism must be initialized before you can make calls of the form just shown. Example 10-9 shows the code that should be placed in the widget's initialize method to initialize this mechanism.

Example 10-9. BitmapEdit: initializing Xmu's atom caching mechanism in the initialize method

```
(void) XmuInternAtom( XtDisplay(new), XmuMakeAtom("NULL") );
```

*The X Protocol defines all the predefined atoms. Therefore, even though XA_TARGETS would be convenient to have as a predefined atom, this can't be arranged with changing the protocol (albeit in a minor way). And the protocol is unlikely to be changed in even minor ways when there are workarounds, as in this case.

10.2.9.2 Converting the Standard Selections

The Xmu routine `XmuConvertStandardSelection` can be used to respond to a TARGETS selection request, as well as to other standard targets defined by Xmu.

Example 10-10 shows the portion of the *convert_proc* for BitmapEdit that handles the standard targets.

This code is adapted from the standard client *xclipboard*, and can be copied almost directly into your widget.

Example 10-10. BitmapEdit: converting standard targets in the convert_proc

```
static Boolean
convert_proc(w, selection, target, type_return, value_return,
        length_return, format_return)
Widget w;
Atom *selection;
Atom *target;
Atom *type_return;
XtPointer *value_return;
unsigned long *length_return;
int *format_return;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int x, y;
    int width, height;
    XSelectionRequestEvent* req = XtGetSelectionRequest(w,
            *selection, (XtRequestId) NULL);

    /* handle all required atoms, and the one that we use */
    if (*target == XA_TARGETS(XtDisplay(cw))) {
        /* TARGETS handling copied from xclipboard.c */
        Atom* targetP;
        Atom* std_targets;
        unsigned long std_length;
        XmuConvertStandardSelection(cw, req->time, selection,
                target, type_return,
                (XtPointer*)&std_targets,
                &std_length, format_return);
        *value_return = XtMalloc(sizeof(Atom)*(std_length + 1));
        targetP = *(Atom**)value_return;
        *length_return = std_length + 1;
        *targetP++ = cw->bitmapEdit.target_atom;
        bcopy((char*)std_targets, (char*)targetP, sizeof(Atom)*std_length);
        XtFree((char*)std_targets);
        *type_return = XA_ATOM;
        *format_return = sizeof(Atom) * 8;
        return(True);
    }
    /* Xt already handles MULTIPLE, no branch necessary */
    else if (*target == cw->bitmapEdit.target_atom) {
         /* handle normal selection - code shown in Example 10-6 */
          .
          .
          .
    }
```

Example 10-10. BitmapEdit: converting standard targets in the convert_proc (continued)

```
    else {
        if (XmuConvertStandardSelection(cw, CurrentTime, selection,
                target, type_return, value_return,
                length_return, format_return))
            return True;
        else {
            XtWarning("bitmapEdit: requestor is requesting\
                    unsupported selection target type.\n");
            return(False);
        }
    }
}
```

Overall, this code handles the TARGETS atom in the first branch, the normal selection target in the second, and any remaining standard atoms and any unknown atoms as two cases in the third branch. For ICCCM-compliant code, you can copy this entire function into your widget and then write just the second branch. Note that branches that successfully provide the requested data return `TRUE`, and those branches that don't return `FALSE`.*

In the first branch you will also need to change the reference to the instance part field that stores the target atom used for selections, `bitmapEdit.target_atom`. If your widget uses a predefined atom or one supported by the Xmu facility, you would reference that atom here instead of the instance part field. If you called `XInternAtom` in `initialize` and stored the result in an instance part field, you specify that here.

Note that `XtGetSelectionRequest` is used to get the time from the `SelectionRequest` event that the owner received before Xt called the *`convert_proc`* function. `XtGetSelectionRequest` was introduced in R4 for this specific reason; it needs to meet the current ICCCM and work around the fact that the *`convert_proc`* is defined without an event argument. (The *`convert_proc`* definition could not be changed because of the required backwards compatibility with R3.)

10.2.9.3 The Clipboard Selection

According to the ICCCM, if a widget or application allows the user to delete a selection, the selection owner code should place the deleted data on the CLIPBOARD selection.† The widget should be prepared to respond to a request for the contents of the CLIPBOARD, much as it does for the PRIMARY selection. The only client that requests data in this way will be *xclipboard*.

Except when a widget asserts ownership of the CLIPBOARD with `XtOwnSelection` in order to place newly deleted data on it, the *xclipboard* client is the owner of this property. When it starts up, *xclipboard* asserts ownership of the CLIPBOARD selection. If it loses the

*The ICCCM also specifies that functions implementing selections must be able to respond to a MULTIPLE target value, which is used to handle selections too large to fit into a single property. However, the necessary handling is done by the Intrinsics. Your procedures do not need to worry about responding to the MULTIPLE target value; a selection request with this target type will be transparently transformed into a series of smaller transfers.

†As mentioned earlier, `XA_CLIPBOARD` is not a predefined atom, and must be handled via the Xmu atom caching mechanism or equivalent code.

selection (which will happen whenever a widget or client has newly deleted the contents of a selection), it obtains the contents of the selection from the new owner, then reasserts its own ownership of the selection.

Clients wishing to restore deleted data should request the contents of the CLIPBOARD, using the same techniques as we've shown for the PRIMARY selection. *xclipboard* will respond to these requests, returning the deleted data.

The use of *xclipboard* allows the value of a selection to survive the termination of the original selection owner.

10.2.10 How Incremental Selection Works

An incremental selection is very similar to an atomic transfer, except that you use different set of routines and procedure types, and Xt calls the procedures you register multiple times instead of just once as in atomic transfers. Incremental transfers and atomic transfers are mutually exclusive—an owner can support both, but if the owner doesn't, a requestor cannot use the incremental routines to request a selection owned atomically, or vice versa.

As in an atomic transfer, the owner and the requestor in an incremental transfer only have to make one call each in order to initiate this multiple-section transfer. Here is the procedure:

- The owner calls `XtOwnSelectionIncremental` in response to the user selecting something. `XtOwnSelectionIncremental` returns `True` or `False` to indicate whether ownership was granted. `XtOwnSelectionIncremental` registers four callbacks:

```
XtConvertSelectionIncrProc convert_proc;
XtLoseSelectionIncrProc lose_selection_proc;
XtSelectionDoneIncrProc transfer_done_proc;
XtCancelConvertSelectionProc cancel_conversion_proc;
```

- The user pastes the selection in a widget, which causes that widget to request the selection with `XtGetSelectionValueIncremental`. `XtGetSelectionValueIncremental` registers one callback:

```
XtSelectionCallbackProc requestor_proc;
```

  This callback receives the selection data. The *requestor_proc* is called once upon delivery of each segment of the selection value. It is called with type `XT_CONVERT_FAIL` when the transfer is aborted—the requestor has the option of keeping or disposing of the partial selection. `XtGetSelectionValuesIncremental` is analogous except that it accepts multiple target/*client_data* pairs which must be received in the *requestor_proc*.

- The request for the selection causes Xt to call the *convert_proc*, which supplies the selection as described for atomic transfers, with the following exceptions. First, *convert_proc* is called repeatedly by Xt to get each segment. Therefore, it must keep track of what segments have been delivered. Second, when the last segment has been delivered, it should store a non-`NULL` value in *value* and zero in *length_return*. Third, the *convert_proc* must be ready to supply an additional

requestor while it is still engaged in transferring segments to one requestor. The *convert_proc* is called with a *request_id* argument that identifies the request. You write the *convert_proc* to maintain a record of which segments have been delivered for each request_id. One easy way to do this is to use the Xlib context manager.

- When each segment is converted, Xt calls the *requestor_proc* (once for each segment), which extracts the data for the requestor and actually pastes it. This function may paste the data chunk-by-chunk or wait until the entire transfer is complete and then paste the entire selection.
- A zero-length segment terminates the transfer and results in Xt calling the *transfer_done_proc*.

This was the process of a successful transfer. All the remaining callback functions registered are for less common occurrences or unsuccessful transfers. The *cancel_callback* is called by Xt on timeout. This means that the transfer is considered complete even though it failed. In this process the owner frees memory allocated for the transfer.

Calls of the *lose_ownership_proc* do not indicate completion of in-progress transfers—these transfers should continue. It just means that the owner lost ownership of the selection, so that the owner should unhighlight what the user selected.

If *transfer_done_proc* is specified, the owner allocates and frees storage. If *transfer_done_proc* is NULL, *convert_proc* must allocate storage using XtMalloc, XtCalloc or XtRealloc, but Xt will free that memory. The owner may use XtDisownSelection to relinquish ownership. (This is the same routine used to relinquish ownership of an atomic selection.)

10.2.11 Miscellaneous Selection Routines

If the user deletes the information selected, the owner should call XtDisownSelection. XtSetSelectionTimeout sets the time within which widgets must respond to one another. This is initially set by the XtNselectionTimeout resource, and defaults to 5 seconds. The selection timeout prevents hanging when the user pastes but the current owner is slow or hung. XtGetSelectionTimeout reads the current selection timeout value. Widgets should not normally require these two calls, since the selection timeout should remain under the user's control.

If the requestor can request more than one target type, such as TARGETS and its normal selection target, it normally does so using separate actions. (Both actions can be invoked by the same triggering event, if desired.) Each action specifies a different target type and a different requestor callback. That way, each requestor callback handles only one type of target.

Beware: there is a danger in this approach. The selection owner might change between the repeated XtGetSelectionValue calls. XtGetSelectionValues (plural) can be used instead if the requestor would like to receive the data in more than one representation. The requestor's single callback function would then be called once for each representation. (The owner's *convert_proc* would also be called once per representation.)

11

Geometry Management

This chapter discusses how composite and constraint widgets manage the layout of widgets, and how to write your own simple composite and constraint widgets.

In This Chapter:

How Composite Management Works 322
- Initial Geometry Negotiation 324
- Basic Core Methods in Composite Widgets 328
- Laying Out Child Widgets 330
- The change_managed Method 332
- XtQueryGeometry and the query_geometry Method 332
- XtMakeGeometryRequest and the geometry_manager Method 333
- The insert_child and delete_child Methods 335

How Constraint Management Works 336

Writing a Constraint Widget 336
- The Core Resource List 336
- The Constraint Resource List 337
- Class Structure Initialization 339
- The Constraint initialize Method 341
- The class_part_init Method 342
- The geometry_manager Method 343
- The resize Method 347
- The Core and Constraint set_values Methods 349
- The change_managed Method 349
- The query_geometry Method 350
- Delaying Geometry Recalculation 351

Compound Widgets 351

Stacking Order 352

11
Geometry Management

Composite and constraint widgets lay out the widgets in your application according to certain rules. You cannot hardcode the position of widgets in an application because X applications can be resized and they must reposition their widgets to take advantage of the available space. Because of the window manager, even the initial size of the application may not be the application's preferred size.

Chapter 3, *More Techniques for Using Widgets*, demonstrated how you can use existing composite and constraint widgets in the application. You can control widget layout rules with resources. However, you may find that no existing composite or constraint widget can be configured with resources to have the layout rules you need. In this case, you will need to write your own composite or constraint widget or modify an existing one. However, before embarking on writing one of these widgets, you should realize that composite and constraint widgets are complex. First investigate the alternatives! Perhaps you can find a composite or constraint widget from another widget set that has the layout characteristics you need. If you determine that you have no alternative but to write your own composite or constraint widget, you should keep it as simple as possible. It is much easier to write a special-purpose widget that handles a limited layout situation than it is to write a general-purpose composite or constraint widget like Box or Form. And even Box and Form are simple as composite and constraint widgets go!

A *composite* widget is defined as any widget that is a subclass of the Xt-defined class Composite. A *constraint* widget is any widget that is a subclass of the Xt-defined class Constraint. Constraint is a subclass of Composite. As you may recall, a composite widget is the simplest kind of geometry-managing widget; it handles all its children equally, or handles each child in a fixed way. For example, the Box widget handles all of its children equally. As an example of a special-purpose composite widget, Section 11.2 describes a composite widget called ScrollBox that manages two scrollbars and a main window. This widget requires that it have exactly three children added in a particular order.

A constraint widget has all the characteristics of composite widgets but maintains configurable data about each child so that it can cater to the needs of each child. By setting the constraint resources of a child of a constraint widget, you configure the constraint widget's layout policy for that child. Constraint widgets are inherently more powerful than composite widgets, but are also more complicated to write. A constraint widget requires all the code of a composite widget, plus code to handle the constraints of each child. Because of this complexity, you should hesitate even further before attempting to write a constraint widget. As

an example of the code for a constraint widget, this section describes the Athena Form widget.

Composite and constraint widgets can be used within widgets as well as within applications. For example, you may recall that the *xbitmap* application was implemented using the BitmapEdit widget and adding scrollbars from the application. We could rewrite the BitmapEdit widget to provide its own scrollbars, controllable through resources. This widget would create a composite widget and three children: BitmapEdit and two Scrollbar widgets. This kind of *compound* widget is actually made up of several widgets, but to the application writer it should act like a single widget. The advantage of this rewrite would be that the application code would become simpler, and the bitmap editor with scrollbars could be easily used in other applications. The fourth major section of this chapter describes how to write a compound widget.

Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*, describes how to write a widget capable of being the parent of gadgets, which are windowless widgets.

It is a good idea to define exactly what is meant by the *geometry* of a widget. The geometry of a widget is its position, its size, and its border width. These are the Core fields `x`, `y`, `width`, `height`, and `border_width`. Border width is included because window positions are measured from the origin of the parent (the top-left corner inside the border of the parent) to the top-left corner outside the border of the child. Therefore, changing the border width of a widget by 1 pixel moves the origin of that widget 1 pixel along both the x and y axes relative to its parent. This concept is shown in Figure 11-1.

Geometry management may also control the stacking order of a group of children (that is, which children appear to be on top). However, control of the stacking order requires more programmer effort than control of geometry, because it isn't completely built into Xt.

11.1 How Composite Management Works

Like simple widgets, the characteristics of composite widgets are defined by their methods. The key methods in a composite widget are the Core methods `resize` and `query_geometry` and the Composite methods `geometry_manager` and `change_managed`. We will discuss these methods first, and then move on to the other methods defined by Composite, `insert_child` and `delete_child`, which are infrequently used.

We'll begin with a summary of what these four most important methods do. In short, they handle interactions with the three generations of widgets involved in direct geometry interactions with a composite widget: the parent of the composite widget, the composite widget itself, and the children of the composite widget.

- The `resize` method moves and resizes the child widgets as necessary to fit within the composite widget's new size.
- The `query_geometry` method supplies a preferred geometry to a widget's parent when the parent calls `XtQueryGeometry`. The parent makes this call in the process of determining a new layout for its children.

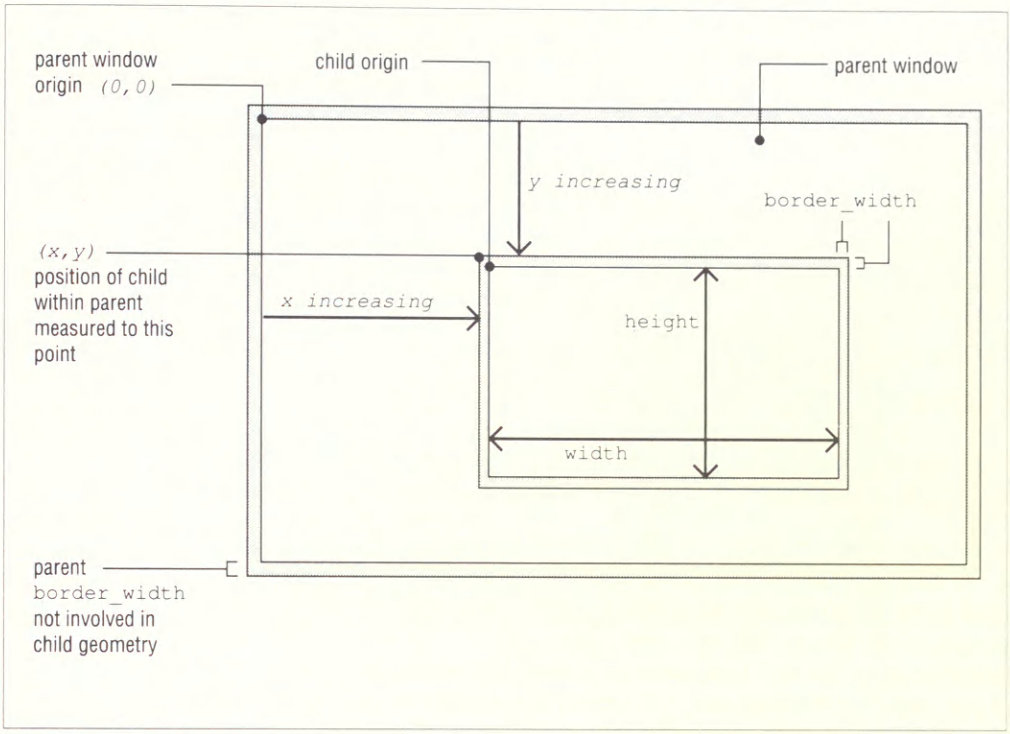


Figure 11-1. Role of border width in widget geometry

- The `geometry_manager` method handles resize requests from the child widgets. Usually, the only kind of child that will make a resize request to the parent is another composite or constraint widget. However, some simple widgets do request resizing. For example, when a Label widget's string is changed through resources, the Label widget increases its own size (this is allowed only in the `set_values` method). Xt then calls the parent's `geometry_manager` method, which must decide whether this new size is acceptable and then make any changes. When a child widget asks for more space and the composite widget doesn't have enough, the composite widget's `geometry_manager` may ask its own parent for more space by calling `XtMakeGeometryRequest`.
- The `change_managed` method changes the layout of the children when a child is managed or unmanaged. A child is managed initially when `XtRealizeWidget` is called (after children are created with `XtCreateManagedWidget`) or when `XtManageChild` or `XtManageChildren` is called to add an already-realized widget to a composite parent's managed set. (A widget can be unmanaged later to remove it from the screen without destroying it, and then managed again at any time. Each time a child is managed or unmanaged, or destroyed, the `change_managed` method is called.)

The `resize` and `query_geometry` methods were already introduced in Chapter 6, *Basic Widget Methods*, as they apply in simple widgets. In composite widgets, they have the same job, but it is more complicated because they now have children to worry about. We will discuss these methods again in this chapter as they appear in composite widgets.

To write any of these four methods, you need to look at all the ways that geometry changes can occur, so that you know all the situations in which the methods will be called. Here are the four most important situations:

- During the negotiation that takes place to determine the initial size of each widget when an application starts up.
- When the user resizes the entire application.
- When a widget requests a size change from the application.
- When a widget is resized by the application.

As you can see, there are many cases. It helps to be systematic about understanding and programming for these cases. The next four sections describe each of these cases one at a time.

The complexity of the geometry negotiations in the following description may be intimidating. A truly general-purpose composite widget is a large, complex piece of software. You should leave this programming to the widget writers that write commercial widget sets, and concentrate on things that are more important in your application. The purpose of this description is to let you understand how complete composite widgets work (not to suggest that you should try to write one). However, it is possible to write small, special-purpose composite widgets that solve particular layout problems. Composite widgets are simpler when they are more authoritarian—when they don't do as much to try to satisfy the preferences of their children. Section 11.2 describes ScrollBox, a simple composite widget designed solely to manage a main widget and two scrollbars. Writing this kind of widget is manageable because the widget manages a fixed number of children and has simple layout rules.

11.1.1 Initial Geometry Negotiation

At least one geometry negotiation takes place in any application, even if the application is never resized. This geometry negotiation occurs when an application starts up.

Figure 11-2 shows the process of initial geometry negotiation in schematic form.

The call to `XtRealizeWidget` initiates a two-step process. When `XtRealizeWidget` is called on the top-level widget, most or all of the widgets have been created but windows have not been created for them. `XtRealizeWidget` initiates a geometry negotiation that ripples through the widget hierarchy (as is described below). The widgets' `realize` methods (which create windows for the widgets) are not called until this process is complete.

`XtRealizeWidget` first calls the `change_managed` method of every composite widget in the application, beginning with the lowest widgets in the hierarchy (called a *post-order* traversal). Each `change_managed` method determines an initial size for each child and calls `XtMoveWidget` and/or `XtResizeWidget` for the child. All the `change_managed` methods are called until the one in the Shell widget. (Remember that

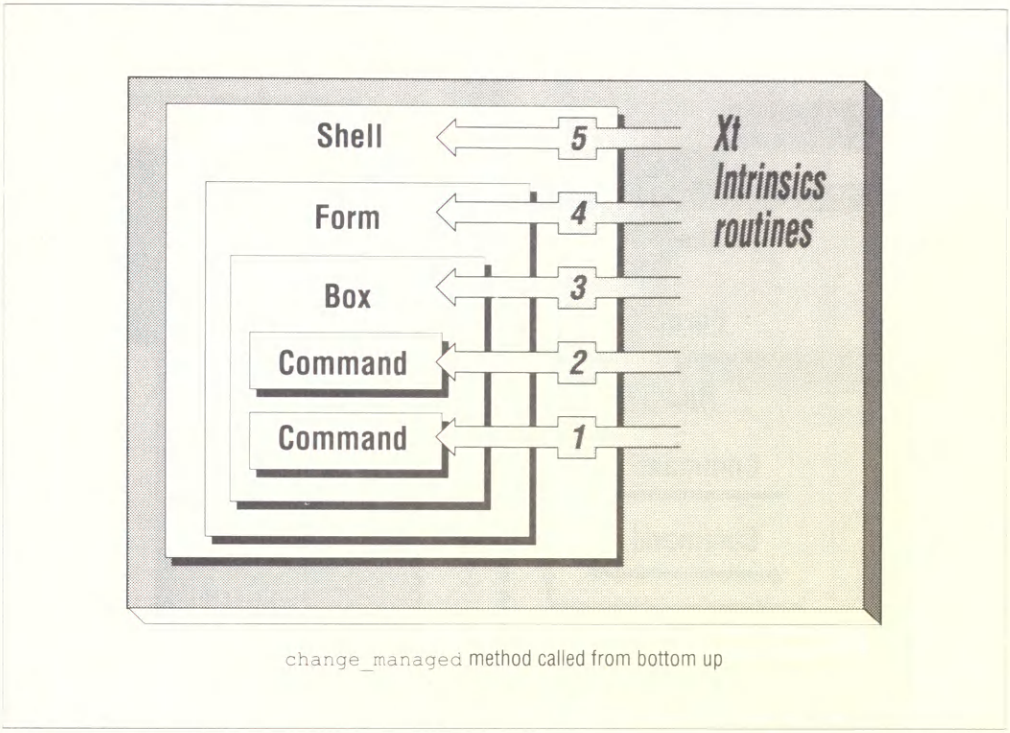


Figure 11-2. Initial geometry negotiation, assuming sufficient shell space

the Shell widget is a composite widget that has only one child.) The child is also a composite widget (except in single-widget applications such as *xhello*). When the Shell widget is reached, the Shell widget size is set to the size of its child and the process stops unless the user has specified an initial geometry for the entire application through the resource database or command line.

A `change_managed` method can (but is not required to) determine each of its children's preferred geometry by calling `XtQueryGeometry` for each child. This will result in the `query_geometry` method of each child being called. Instead of calling `XtQueryGeometry`, the `change_managed` method may use the child's Core `width` and `height` fields.

The `change_managed` method does not determine the composite widget's own size. That job is for the parent of the composite widget, which is another composite widget.

If the user-specified, top-level widget geometry is different from the geometry of the Shell widget's child after all the `change_managed` methods are called, then the Shell widget resizes its child to the user-specified size. This makes Xt call the `resize` method of the child composite, and this `resize` method reconsiders the layout of its children. This process proceeds down the chain of widgets to the bottom. At each stage the `resize` method can (but need not) call `XtQueryGeometry` for each child to get each child's opinion of the intended geometry for that child.

Figure 11-3 shows the continued process of initial geometry negotiation if the user has specified the top-level geometry through resources rather than accepting the application's built-in defaults.

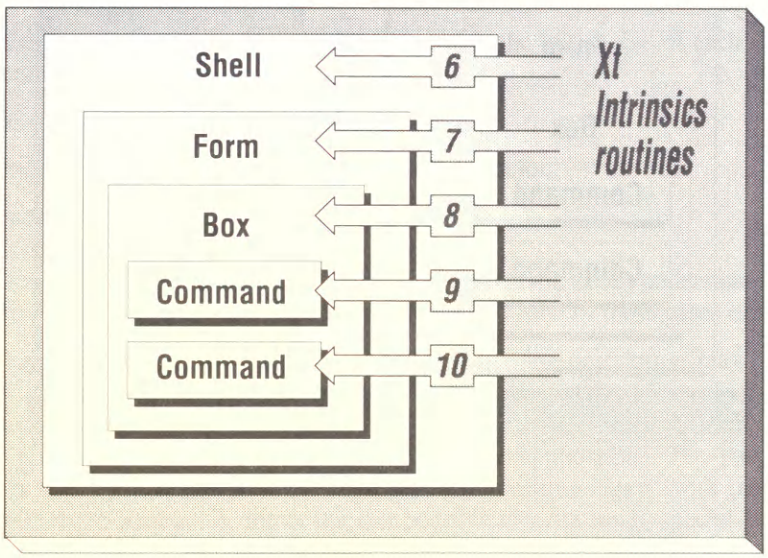


Figure 11-3. Initial geometry negotiation, if resizing is necessary

Note that this process and the methods involved are more complicated Therefore, we do not need a `geometry_manager` method.

A ScrollBox widget is shown managing BitmapEdit and two scrollbars in Figure 11-4.*

The Athena Viewport widget does scrollbar management in a more general way than does ScrollBox. It is a subclass of Form that takes any main window as a child and creates scrollbars. It shows only a small portion of the main window and uses the scrollbars to determine which portion of the main window is shown. But Viewport doesn't work well with BitmapEdit because BitmapEdit has a built-in ability to display in a smaller window that conflicts with Viewport's efforts. Besides, Viewport is several times larger and more complicated than ScrollBox, because it includes the scrollbar callback functions and because it honors a child's geometry preferences. ScrollBox is a modest widget that manages the geometry

*ScrollBox widens the borders of its children, for an unknown reason. It might as well be admitted that there is probably a bug in it somewhere!

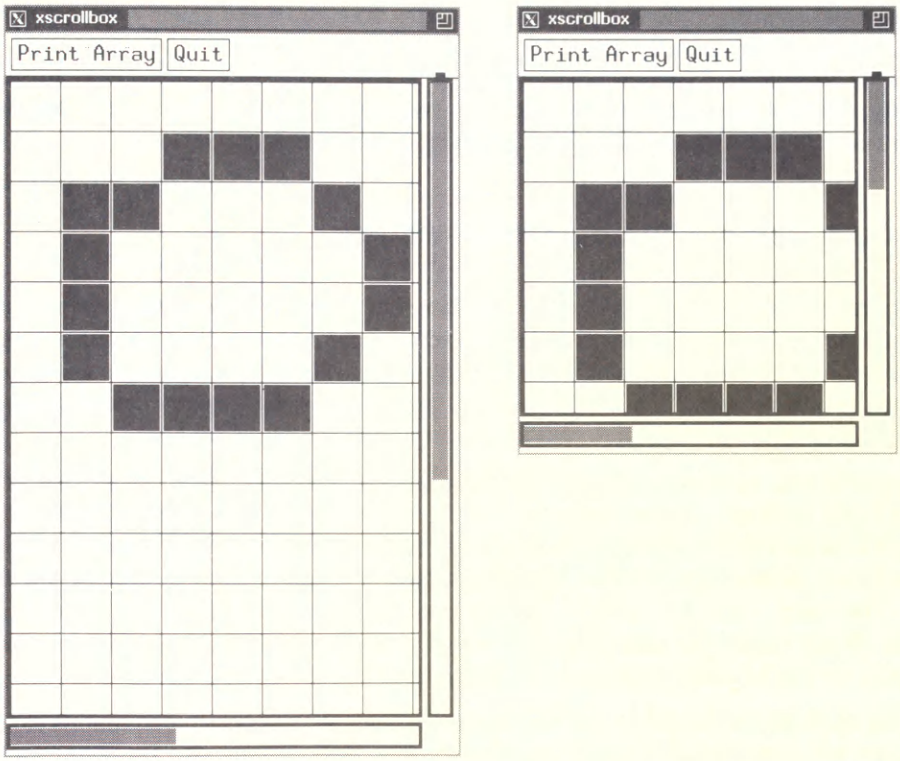


Figure 11-4. A ScrollBox widget at two different sizes

of scrollbars, leaving their connection with the main window up to the application. This demonstrates the essential elements of a composite widget without too much complication.*

The ScrollBox widget code, along with a version of *xbitmap* that uses it, is available in the example source in the *ch09* directory. It lays out its children by adjusting the width and height of the three children so that they fill the ScrollBox widget, while keeping the width of the Scrollbar widgets constant. The width of the Scrollbar widgets is set through their resources and is never modified by ScrollBox.

The first few sections below describe the methods that are used in ScrollBox and that are required in all composite widgets. First, we discuss the Core `initialize`, `realize`, `set_values`, `resize`, and `query_geometry` methods as they are used in composite widgets. Then, we discuss how the ScrollBox widget implements layout calculations in a

*It is worth noting that the Box widget fails miserably in managing scrollbars, while Form is adequate but has the annoying characteristic that it resizes the width of the scrollbars as well as their length, sometimes resulting in bloated or miniscule scrollbars.

common routine called by the set_values, resize, and change_managed values. This is followed by further discussion of the change_managed and query_geometry methods. Then, we go on to discuss the methods not used by ScrollBox, but that would be used in more complicated composite widgets, in particular geometry_manager. Finally, we briefly discuss the methods available in composite widgets but rarely needed: set_values_almost, insert_child, and delete_child.

11.1.2 Basic Core Methods in Composite Widgets

Both composite and constraint widgets are subclasses of Core. Therefore, they have all the Core methods described in Chapters 5 and 6. However, since composite and constraint widgets usually have no input and output semantics, the expose method is set to NULL and the widget has no default translation table or actions. As a result, all the event-oriented fields in the Core class structure become irrelevant to composite and constraint widgets.

But composite and constraint widgets do use the Core initialize, realize, and set_values methods. These methods have the same roles as for simple widgets. The initialize method initializes instance part variables and checks initial resource values. The realize method sets window attribute values and then creates a window for the widget. The set_values method updates any instance part fields that depend on resources. Since composite and constraint widgets don't need GCs, initialize and set_values don't contain code to create and change GCs as in simple widgets.

These three methods for ScrollBox are absolutely minimal, and call a common routine called DoLayout when any actual sizing or positioning of widgets is required. The initialize method simply sets the widget's default width and height, the realize method is inherited, and the set_values method changes the layout of children when either of ScrollBox's two resources is changed. These resources control the vertical and horizontal distance in pixels that will be left between the Scrollbar widgets and the main widget, and between each of these widgets and the borders of ScrollBox. Example 11-1 shows the set_values method of ScrollBox.

Example 11-1. ScrollBox: the set_values method

```
/* ARGSUSED */
static Boolean SetValues(current, request, new, args, num_args)
Widget current, request, new;
ArgList args;
Cardinal *num_args;
{
    ScrollBoxWidget sbwcurrent = (ScrollBoxWidget) current;
    ScrollBoxWidget sbwnew = (ScrollBoxWidget) new;

   /* need to relayout if h_space or v_space change */
    if ((sbwnew->scrollBox.h_space != sbwcurrent->scrollBox.h_space) ||
            (sbwnew->scrollBox.v_space !=
            sbwcurrent->scrollBox.v_space))
        DoLayout(sbwnew);

    return False;
}
```

Two more Core methods are used in composite widgets: `resize` and `query_geometry`. The `resize` method changes the layout of its children and is shown in Example 11-2.

Example 11-2. ScrollBox: the resize method

```
static void Resize(w)
Widget w;
{
    ScrollBoxWidget sbw = (ScrollBoxWidget) w;

    DoLayout(sbw);
}
```

The `query_geometry` method answers the parent's inquiry about a size change for this composite widget and is shown in Example 11-3.

Example 11-3. ScrollBox: the query_geometry method

```
/* ARGSUSED */
static XtGeometryResult QueryGeometry(w, request, reply_return)
Widget w;
XtWidgetGeometry *request, *reply_return;
{
    XtGeometryResult result;

    request->request_mode &= CWWidth | CWHeight;

    if (request->request_mode == 0)
    /* parent isn't going to change w or h, so nothing to
     * re-compute */
       return XtGeometryYes;

    /* if proposed size is large enough, accept it.  Otherwise,
     * suggest our arbitrary initial size. */

    if (request->request_mode & CWHeight) {
        if (request->height < INITIAL_HEIGHT) {
            result = XtGeometryAlmost;
            reply_return->height = INITIAL_HEIGHT;
            reply_return->request_mode &= CWHeight;
        }
        else
            result = XtGeometryYes;
    }

    if (request->request_mode & CWWidth) {
        if (request->width < INITIAL_WIDTH) {
            result = XtGeometryAlmost;
            reply_return->width = INITIAL_WIDTH;
            reply_return->request_mode &= CWWidth;
        }
        else
            result = XtGeometryYes;
    }

    return(result);
}
```

Although the query_geometry method has the same role in all widgets, composite and simple, a composite widget's size preference depends on its children. Normally this means the query_geometry method will query its children and try different layouts until it arrives at the geometry, or some approximation of it, suggested by its parent. This calculation is complicated because the widget may have any kind of child, and their responses to geometry suggestions are unpredictable. ScrollBox ignores this complexity because it knows exactly what kinds of children it will have and what their characteristics are. Therefore, its query_geometry method is basically the same as the query_geometry method of a simple widget.

To be more precise, what this query_geometry method does is accept any size suggested by the parent which is larger than the minimum useful size of the application. When the suggested size is too small, the query_geometry method uses the minimum useful size as a compromise. Note, however, that this is really hardcoding the characteristics of the child into our composite widget. It would be better to add resources to control the minimum useful size.

11.1.3 Laying Out Child Widgets

Composite widgets need to calculate a layout and manipulate their child widgets from set_values, from resize, and from change_managed. Therefore, in most composite widgets this common code is placed in a single routine called DoLayout. Example 11-4 shows the DoLayout routine from ScrollBox.

Example 11-4. ScrollBox: private routine to lay out child widgets

```
/* ARGSUSED */
static DoLayout(w)
Widget w;
{
    ScrollBoxWidget sbw = (ScrollBoxWidget) w;
    Widget main, vscroll, hscroll;
    Widget child;
    Dimension mw, mh; /* main window */
    Dimension vh;     /* vertical scrollbar length (height) */
    Dimension hw;     /* horizontal scrollbar length (width) */
    Position vx;
    Position hy;
    int i;

    if (sbw->composite.num_children != 3)
        XtAppError(XtWidgetToApplicationContext(sbw),
                "ScrollBox: must manage exactly three widgets.");

    for (i = 0; i < sbw->composite.num_children; i++) {
        child = sbw->composite.children[i];
        if (!XtIsManaged(child)) {
            XtAppError(XtWidgetToApplicationContext(sbw),
                "ScrollBox: all three widgets must be managed.");
        }
    }
```

Example 11-4. ScrollBox: private routine to lay out child widgets (continued)

```
    /* Child one is the main window, two is the vertical scrollbar,
     * and three is the horizontal scrollbar.  */

    main = sbw->composite.children[0];
    vscroll = sbw->composite.children[1];
    hscroll = sbw->composite.children[2];

    /* Size all three widgets so that space is fully utilized. */

    mw = sbw->core.width - (2 * sbw->scrollBox.h_space) -
            vscroll->core.width - (2 * vscroll->core.border_width) -
            (2 * main->core.border_width);

    mh = sbw->core.height - (2 * sbw->scrollBox.v_space) -
            hscroll->core.height - (2 * hscroll->core.border_width) -
            (2 * main->core.border_width);

    vx = main->core.x + mw + sbw->scrollBox.h_space +
            main->core.border_width + vscroll->core.border_width;
    hy = main->core.y + mh + sbw->scrollBox.v_space +
            main->core.border_width + hscroll->core.border_width;

    vh = mh;    /* scrollbars are always same length as main window */
    hw = mw;

    XtResizeWidget(main, mw, mh);

    XtResizeWidget(vscroll, vscroll->core.width, vh);
    XtMoveWidget(vscroll, vx, vscroll->core.y);

    XtResizeWidget(hscroll, hw, hscroll->core.height);
    XtMoveWidget(hscroll, hscroll->core.x, hy);
}
```

In general, `DoLayout` moves and resizes the child widgets according to its layout policy. This routine may query the children with `XtQueryGeometry` before making decisions, but it is not required to. In this case, there is no need to because ScrollBox handles only two types of widgets with no size preferences.

`DoLayout` is passed only one argument, ScrollBox's own widget ID (a pointer to its widget instance structure). But the composite `children` field in ScrollBox's instance structure is an array of the IDs of all the children, and `num_children` is the number of children.*

When each child is added to a composite widget, its ID is added to the `children` field of the composite part of the instance structure, and the `num_children` field is incremented. Therefore, the code to lay out the children is usually a loop that treats each child one at a time. This often takes two passes, since the routine needs to know which children are managed before it can determine their final geometries. All children, even unmanaged ones, are listed in the `children` and `num_children` fields.

This particular `DoLayout` procedure makes sure that there are exactly three children and that they are all managed. Then, it calculates the positions and sizes for all the children so that they will fill all the available space in ScrollBox's own window. Finally, it calls

*Incidentally, the `children` and `num_children` fields are resources. However, they are read-only from outside the widget code; the application should never set them with `XtSetValues`.

XtResizeWidget and XtMoveWidget, which check to see if there was any change before making Xlib calls to move and resize the windows.

11.1.4 The change_managed Method

In every composite widget, the change_managed method is called once (and only once, even when there are multiple children) during the XtRealizeWidget process to determine an application's initial layout. change_managed is also called when an application later unmanages a managed widget or manages an unmanaged widget (as long as the XtNmappedWhenManaged resource has its default value). Therefore, change_managed also calls DoLayout.

An application unmanages a widget to remove the widget from visibility without destroying it, and at the same time to tell the composite widget to change the layout of the remaining widgets to fill the gap. This is done by calling XtUnmanageChild or XtUnmanageChildren. The application can then make the composite widget redisplay the widget by calling XtManageChild or XtManageChildren. This response depends on the Core XtNmappedWhenManaged resource having its default value, TRUE. When set to FALSE, the management state has no effect on mapping, and the application must call XtMapWidget and XtUnmapWidget instead. Usually an application does this so that a widget will become invisible *without* triggering a re-layout to fill in the space it has vacated. Therefore, change_managed need not check the XtNmapWhenManaged resource of each child.

You have now seen all of the code of ScrollBox! To summarize, a very basic composite widget such as ScrollBox has a standard initialize method, resize and change_managed methods that just call DoLayout, and a set_values method that calls DoLayout when any resource that affects layout is changed. The DoLayout routine actually lays out the children. The widget's query_geometry method is basically just like a simple widget's query_geometry. Now we'll move on to describe what may be added to this skeleton to make more fully-featured composite widgets.

11.1.5 XtQueryGeometry and the query_geometry Method

We have mentioned that XtQueryGeometry just calls a child's query_geometry method, but not the details of how this works. The query_geometry method for simple widgets is described in Chapter 6, *Basic Widget Methods*. The role of this method in composite widgets is the same, but the details of its job are different. You may recall that this method is passed pointers to two XtWidgetGeometry structures, one which specifies the parent's proposed geometry, and the other which is used by the child to return a compromise geometry. These two structures are allocated by the method that calls XtQueryGeometry and passed as pointers to that call. The XtGeometryResult enum returned by the query_geometry method is passed right through as the returned value of XtQueryGeometry.

Composite and constraint widgets play the role of both parent and child. When you write a composite widget, you may call XtQueryGeometry in several places to get the child's response to your proposed size. You will also need to write a query_geometry method so that your composite can respond to its parent's XtQueryGeometry request.

A query_geometry method in a composite widget should base its response on the size preferences of its children. It should calculate a new layout based on the proposed geometry passed in, and then query its children to get their opinions of their new geometry. If any of the children is a composite widget, they may query their children, and so on. Therefore, these requests tend to trickle down to the lowest widget in the hierarchy. ScrollBox took the biggest shortcuts in its query_geometry method. Not only didn't it query its children, but it hardcoded its response based on the characteristics of the kind of main window it expected. This would be the first place to begin improving ScrollBox.

Note, however, that a composite widget is allowed to be authoritarian and not ask its children whether they like the sizes they are about to be given. However, this kind of composite widget will not be suitable as a parent of a widget that really needs certain size preferences.

In R4 and later, a parent must specify a complete proposed geometry when calling XtQueryGeometry, not just the changes it intends to make as was specified in R3.

11.1.6 XtMakeGeometryRequest and the geometry_manager Method

XtMakeGeometryRequest calls are made for two reasons. First, when a composite widget honors its children's size preferences, it may find that its current size is inadequate to lay out its children. In this case, it should ask its parent to be resized by calling XtMakeGeometryRequest. Second, Xt calls XtMakeGeometryRequest for a widget when the application has changed a resource that affects geometry.

As mentioned above, XtMakeGeometryRequest calls the parent's geometry_manager method. The parent's geometry_manager has the job of deciding whether the size proposed by the child is acceptable. A subclass of Composite must either define a geometry_manager method, or set this field in the class structure to NULL, because there is no default method to inherit. The XtInheritGeometryManager symbol can be used only in subclasses of a class that defines a geometry_manager method. Any composite widget allowing its children to suggest resizing will require a geometry_manager method of its own.

The way the arguments and returned values are passed between XtMakeGeometryRequest and the parent's geometry_manager method is almost exactly parallel to the way XtQueryGeometry calls the child's query_geometry method. Both calls take pointers to two structures of the same types where one is used for a returned compromise. Both take no more arguments other than the widget ID. Both return an enum value of type XtGeometryResult. The returned value of the geometry_manager method is, generally speaking, passed through as the returned value of XtMakeGeometryRequest. Review Section 6.6 so that these structures, their fields and values, and the returned values are fresh in your mind.

One difference between the way the query_geometry and geometry_manager methods are invoked is that the geometry_manager method can return a fourth enum value, XtGeometryDone (in addition to XtGeometryYes, XtGeometryNo, and XtGeometryAlmost). The return codes of the geometry_manager method are summarized in Table 11-1. set_values_almost is to accept the compromise geometry proposed by the parent or to propose a different geometry to the parent. Once a new geometry is proposed by the set_values_almost method, Xt calls the parent's geometry_manager method again, and the cycle repeats until the geometry_manager returns XtGeometryYes or XtGeometryDone, or until the child gives up trying to change size. Figure 11-5 illustrates this process.

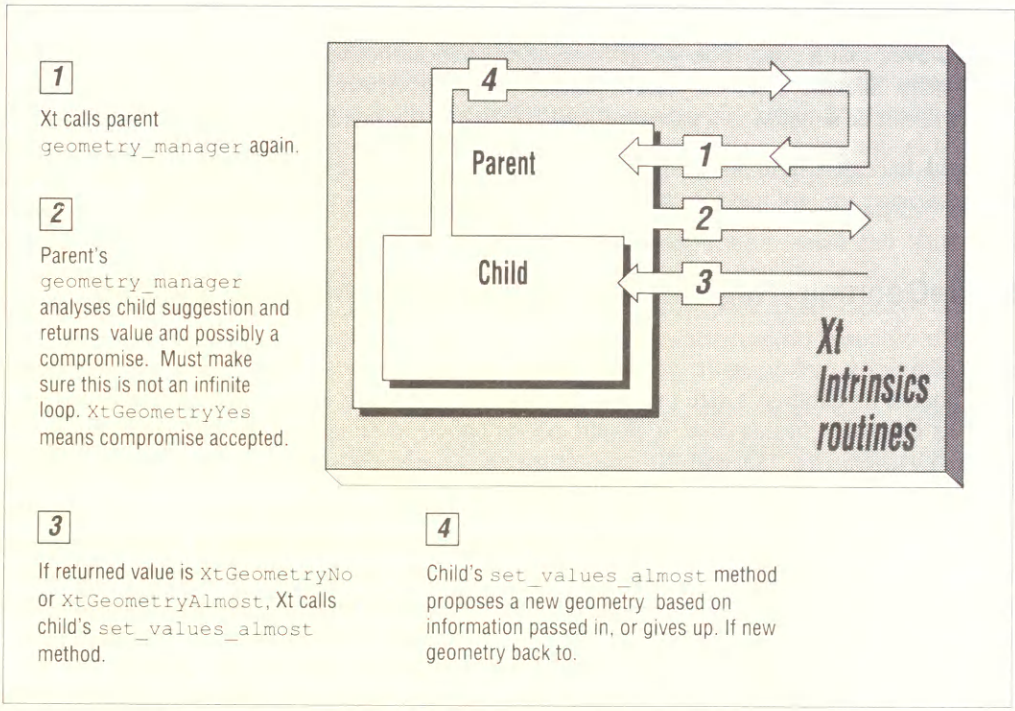


Figure 11-5. Geometry negotiation by the set_values_almost method

Most widgets inherit this method from the Core widget by specifying XtInheritSetValuesAlmost in the Core class part initialization. This inherited method always approves the suggestion made by the parent geometry_manager method. If your widget really depends on being certain sizes, however, you will need to write a set_values_almost method. You should never specify a NULL set_values_almost method because Xt will print a warning message when set_values_almost would have been called, and continue as if it had been called and had returned XtGeometryYes.

The set_values_almost method is passed pointers to two XtWidgetGeometry structures: request and reply. The request structure contains the child's original request and reply includes the geometry_manager method's compromise geometry if geometry_manager returned XtGeometryAlmost. To accept the compromise, the procedure must copy the contents of the reply geometry into the request geometry; to attempt an alternate geometry, the procedure may modify any part of the request argument; to terminate the geometry negotiation and retain the original geometry, the procedure must set request->request_mode to zero.

If geometry_manager returned XtGeometryNo, it will not have generated a compromise. In this case, the set_values_almost method may suggest a new geometry, but it is probably not worth it since the method has no information upon which to base its changes to its previous suggestion. The set_values_almost method at this point should usually just set request->request_mode to zero to terminate the geometry negotiation.

11.1.7 The insert_child and delete_child Methods

The Composite class has an instance part structure that contains an array of all the widget's children (even those not currently managed), the current number of children, and the total number of child slots available. The insert_child method inserts the ID of a child into this array. It is called when the child is created by a call to XtCreateWidget or XtCreateManagedWidget. Most widgets inherit the insert_child method from the Composite class by specifying the symbolic constant XtInheritInsertChild in the class structure initialization. A class would replace the default insert_child method to control the position of each child added, or to limit the number or classes of widgets that can be added.

A composite widget can control the position of each child added by calling a function whose pointer is stored in the instance part field insert_position. The function should return the number of widgets before the widget. The XtNinsertPosition resource sets this function pointer. The default insert_position function returns the current number of children. Of course, because this resource's value is a function pointer, it can be specified in the application only at run time, never through the resource files or command line.

The delete_child method removes the ID of a child from the child array and is called when the application calls XtDestroyWidget. This method is almost always inherited from Composite by specifying the symbolic constant XtInheritDeleteChild in the class structure initialization.

11.2 How Constraint Management Works

The first thing to realize about constraint widgets is that everything said about composite widgets is still true. Because Constraint is a subclass of Composite, all the methods described above are still present and have the same tasks. However, constraint widgets also maintain a structure full of data attached to each child, set through resources. Every time it lays out the children, the constraint widget reads this data to determine how to handle that child. Of course, it still may query each children to get its opinion of a new size. The constraint information adds another level of complexity to the situation.

Like composite widgets, constraint widgets can be drastically simplified by reducing flexibility and features. The Athena Form widget, for example, never queries its children for their geometry input and never asks its parent for a size change. Furthermore, its constraints for each child are quite limited. This makes Form quite short and simple, but also means that it doesn't always do the right thing.

11.3 Writing a Constraint Widget

The following sections describe the portions of the Athena Form widget that relate to geometry management. This will give you a birds-eye view of constraints in action.

11.3.1 The Core Resource List

The Form widget has only one resource of its own, `XtNdefaultDistance`, as shown in Example 11-5. This resource is used only as the default for two of the Constraint resources, `XtNhorizDistance` and `XtNvertDistance`. `XtNdefaultDistance` is used to set the instance field `default_spacing`, which is used in only one place in the widget, in the Constraint `initialize` method described in Section 11.4.4.

Example 11-5. Form: the Core resource list

```
#define Offset(field) XtOffsetOf(FormRec, form.field)
static XtResource resources[] = {
    {
        XtNdefaultDistance,
        XtCThickness,
        XtRInt,
        sizeof(int),
        Offset(default_spacing),
        XtRImmediate,
        (XtPointer)4
    }
};
#undef Offset
```

11.3.2 The Constraint Resource List

The Form widget has three groups of constraint resources. `XtNhorizDistance`, `XtNfromHoriz`, `XtNvertDistance`, and `XtNfromVert` together control the initial position of a child. `XtNtop`, `XtNleft`, `XtNbottom`, and `XtNright` govern repositioning of the child when Form is resized. The `XtNresizable` resource controls whether the `geometry_manager` of this widget will honor requests to change the geometry of this child. Note that `XtNresizable` does not control whether this constraint widget can resize a child—only whether or not it will do so because of a request from the child.*

For more details about how these constraint resources work, read about them on the reference page for the Form widget in Volume Five, *X Toolkit Intrinsics Reference Manual*.

Constraint resources are also called simply *constraints*, particularly because they are stored in a Core instance field called `constraints`. Example 11-6 shows Form's constraint resource list.

Example 11-6. Form: constraint resource list

```
static XtEdgeType defEdge = XtRubber;

#define Offset(field) XtOffsetOf(FormConstraintsRec, form.field)
static XtResource formConstraintResources[] = {
    {
    XtNhorizDistance,
    XtCThickness,
    XtRInt,
    sizeof(int),
    Offset(dx),
    XtRImmediate,
    (XtPointer)DEFAULTVALUE
    },
    {XtNfromHoriz, XtCWidget, XtRWidget, sizeof(Widget),
        Offset(horiz_base), XtRWidget, (XtPointer)NULL},
    {XtNvertDistance, XtCThickness, XtRInt, sizeof(int),
        Offset(dy), XtRImmediate, (XtPointer)DEFAULTVALUE},
    {XtNfromVert, XtCWidget, XtRWidget, sizeof(Widget),
        Offset(vert_base), XtRWidget, (XtPointer)NULL},

    {XtNtop, XtCEdge, XtREdgeType, sizeof(XtEdgeType),
        Offset(top), XtREdgeType, (XtPointer)&defEdge},
    {XtNbottom, XtCEdge, XtREdgeType, sizeof(XtEdgeType),
        Offset(bottom), XtREdgeType, (XtPointer)&defEdge},
    {XtNleft, XtCEdge, XtREdgeType, sizeof(XtEdgeType),
        Offset(left), XtREdgeType, (XtPointer)&defEdge},
    {XtNright, XtCEdge, XtREdgeType, sizeof(XtEdgeType),
        Offset(right), XtREdgeType, (XtPointer)&defEdge},

    {XtNresizable, XtCBoolean, XtRBoolean, sizeof(Boolean),
        Offset(allow_resize), XtRImmediate, (XtPointer)FALSE},
};
#undef Offset
```

*The fact that Form does not provide individual control over the resizability of each child is a major weakness.

The corresponding data structure that this resource list references, FormConstraints, is defined in the private include file for the widget. Its definition is shown in Example 11-7.

Example 11-7. Form: constraint data structure

```
typedef struct _FormConstraintsPart {
/*
 * Constraint Resources.
 */
    XtEdgeType  top, bottom,     /* where to drag edge on resize     */
                left, right;
    int         dx;              /* desired horiz offset             */
    int         dy;              /* desired vertical offset          */
    Widget      horiz_base;      /* measure dx from here if non-null */
    Widget      vert_base;       /* measure dy from here if non-null */
    Boolean     allow_resize;    /* TRUE if child may request resize */

/*
 * Private constraint variables.
 * These store the dimensions of the child prior to layout.
 */

    int         virtual_width, virtual_height;

/*
 * Size of this child as it would be if we did not impose the
 * constraint that its width and height must be greater than zero (0).
 */

    LayoutState layout_state;    /* temporary layout state   */
} FormConstraintsPart;

typedef struct _FormConstraintsRec {
    FormConstraintsPart form;
} FormConstraintsRec, *FormConstraints;
```

The constraints part structure should be considered an instance part structure. This structure has public fields set through resources and private fields that hold state data, just like an instance part structure. Note also that the FormConstraints structure is built the same way as instance structures, by combining part structures for each class into a complete constraint structure. This allows subclasses of Form to create their own constraint part structure and add it after the Form constraint part.

When a widget is created as a child of a constraint widget, the constraint instance structure (FormConstraintsRec, in this case) is placed in the constraints field of the Core instance structure. Xt makes the constraint resources stored there settable, like resources defined by the child even though they are actually defined and used by the parent.

Note that the constraint resource list of a widget can be queried with XtGetConstraintResourceList, although this is rarely needed in widget or application code.

11.3.3 Class Structure Initialization

The Form class is a subclass of Constraint. Therefore, its class structure contains class parts for Core, Composite, Constraint, and Form. Example 11-8 shows the class structure initialization of Form. Several methods referenced here have not been discussed so far in this book. They are the Core methods `class_initialize` and `class_part_init`, and the Constraint methods `initialize` and `set_values`. These and all the geometry management-related methods of Form will be discussed in Section 11.4.6.

Example 11-8. Form: class structure initialization

```
FormClassRec formClassRec = {
  { /* Core class fields  */
    /* superclass         */    (WidgetClass) &constraintClassRec,
    /* class_name         */    "Form",
    /* widget_size        */    sizeof(FormRec),
    /* class_initialize   */    ClassInitialize,
    /* class_part_init    */    ClassPartInitialize,
    /* class_inited       */    FALSE,
    /* initialize         */    Initialize,
    /* initialize_hook    */    NULL,
    /* realize            */    XtInheritRealize,
    /* actions            */    NULL,
    /* num_actions        */    0,
    /* resources          */    resources,
    /* num_resources      */    XtNumber(resources),
    /* xrm_class          */    NULLQUARK,
    /* compress_motion    */    TRUE,
    /* compress_exposure  */    TRUE,
    /* compress_enterleave*/    TRUE,
    /* visible_interest   */    FALSE,
    /* destroy            */    NULL,
    /* resize             */    Resize,
    /* expose             */    XtInheritExpose,
    /* set_values         */    SetValues,
    /* set_values_hook    */    NULL,
    /* set_values_almost  */    XtInheritSetValuesAlmost,
    /* get_values_hook    */    NULL,
    /* accept_focus       */    NULL,
    /* version            */    XtVersion,
    /* callback_private   */    NULL,
    /* tm_table           */    NULL,
    /* query_geometry     */    PreferredGeometry,
    /* display_accelerator*/    XtInheritDisplayAccelerator,
    /* extension          */    NULL
  },
  { /* Composite class fields */
    /* geometry_manager   */   GeometryManager,
    /* change_managed     */   ChangeManaged,
    /* insert_child       */   XtInheritInsertChild,
    /* delete_child       */   XtInheritDeleteChild,
    /* extension          */   NULL
  },
  { /* Constraint class fields */
    /* subresourses       */   formConstraintResources,
    /* subresource_count  */   XtNumber(formConstraintResources),
```

Example 11-8. Form: class structure initialization (continued)

```
    /* constraint_size      */   sizeof(FormConstraintsRec),
    /* initialize           */   ConstraintInitialize,
    /* destroy              */   NULL,
    /* set_values           */   ConstraintSetValues,
    /* extension            */   NULL
  },
  { /* Form class fields    */
    /* layout               */   Layout
  }
};
WidgetClass formWidgetClass = (WidgetClass)&formClassRec;
```

Note that the Form class is the first widget we have shown that defines a class part field—a method of its own, called `layout`. Since this method is not known to Xt, Xt will never call it. The widget must invoke this method itself at the appropriate times (you will see this invocation in the methods below). This code is made into a method instead of just a private function only to make it possible for subclasses of this widget to inherit or replace the method. Having such a method requires that the widget have a `class_part_init` method to handle the inheritance if a subclass specifies the `layout` method with the symbolic constant `XtInheritLayout` (also defined in this class's private header file).

Section 11.2.1 described which Core and Composite methods are required for composite widgets, and how to initialize the other Core and Composite fields for a composite widget. The same is true for constraint widgets.

However, the Constraint part is probably new to you. The `ConstraintClassPart` structure contains seven fields. The first three fields are where the constraint resource list, the number of resources, and the size of the constraint instance structure are entered. This resource list and instance structure were described in the last section. These fields are analogous to the `resources`, `num_resources`, and `widget_size` fields in the Core class part.

The three next fields, `initialize`, `destroy`, and `set_values` are methods defined by the Constraint class. These methods have the same field names as methods of Core, but are fields of a different structure, and contain pointers to different functions that you may need to write. To differentiate Constraint methods from the Core methods, we will precede the names of Constraint fields with the word "Constraint" and the names of Core fields with the word "Core" throughout this chapter.

Two of the three Constraint methods will be described where they fit in below. We'll describe one of them, Constraint `destroy`, now, because it is not used in Form and is less likely to be needed in the constraint widgets you may write. The Constraint `destroy` method is called when a child is destroyed, just before the Core `destroy` method of the child. It is responsible for freeing any memory allocated by the constraint widget that was used to manage that child. However, like the Core `destroy` method, it does not need to free memory allocated by Xt, such as the constraint data structure for the child.

11.3.4 The Constraint initialize Method

The Constraint `initialize` method is called when a widget is created, soon after the Core `initialize` method. It has the same two responsibilities as the Core initialize method, and one additional responsibility. It must:

- Validate the ranges of resource settings, since they may be user-supplied.
- Compute the value of any private constraint instance part fields that depend on constraint resource values (public constraint instance part fields).
- Set child Core geometry fields to match the constraint resources. For example, if a constraint for the maximum height of a widget is set and the initial value set by the child is larger, the Constraint `initialize` method resets the height field in the Core instance structure.

However, like the Core `initialize` method, the Constraint `initialize` method is responsible only for constraint resources and for Core geometry resources. It need not handle any resources of superclasses (other than the Core geometry resources).

The Form widget performs only one of the tasks listed above, initializing constraint resources. In Form's case, the Constraint `initialize` method (shown in Example 11-9) simply sets the initial values of the `XtNvertDistance` and `XtNhorizDistance` constraint resources to the current value of the `XtNdefaultDistance` Form resource, unless the user has specified a value for either constraint resource. This is done only so that the application can set the Form resource once and have it apply to every child that does not override the value.

Form doesn't validate the values of any user-supplied resource values as it should. For example, the user may supply a negative value for the `XtNhorizDistance` or `XtNvertDistance` resources. This would certainly make the layout look bad, but it could also cause the Form widget to go into an infinite loop on geometry negotiations. In general, all `initialize` methods in Core and Constraint should check for ranges of reasonable values of resources where this makes sense. Range checking eliminates a potential source of bugs. Range checking in `set_values` is also a good idea to give the programmer good warning messages).

Example 11-9. Form: the Constraint initialize method

```
#define DEFAULTVALUE -9999

/* ARGSUSED */
static void ConstraintInitialize(request, new)
    Widget request, new;
{
    FormConstraints form = (FormConstraints)new->core.constraints;
    FormWidget fw = (FormWidget)new->core.parent;

    form->form.virtual_width = (int) new->core.width;
    form->form.virtual_height = (int) new->core.height;

    if (form->form.dx == DEFAULTVALUE)
        form->form.dx = fw->form.default_spacing;
```

Example 11-9. Form: the Constraint initialize method (continued)

```
    if (form->form.dy == DEFAULTVALUE)
        form->form.dy = fw->form.default_spacing;
}
```

Note that the Constraint instance part structure (`FormConstraints`) and the Form widget instance structure (`FormWidget`) are accessed by casting two different fields of the child's instance structure passed in.

The Constraint `initialize` method and the child's Core `initialize` are passed the same two copies of the child's instance structure: `request`, and `new`. The `request` widget is the widget as originally requested. The `new` widget starts with the values in the request, but it has already been updated by calling all superclass `initialize` methods.

11.3.5 The class_part_init Method

The `class_part_init` method should be present in a class that defines new methods in its class part structure. These new methods will never be called by Xt since Xt has no knowledge of when to call them. They can only be invoked directly from the widget code. The purpose of making them methods instead of just functions is to allow subclasses to inherit or replace the functions. The `class_part_init` method actually resolves this inheritance by setting each method field to the pointer provided by this class (the subclass is inheriting the method) or to the pointer provided by the subclass (the subclass is replacing the method). Example 11-10 shows a `class_part_init` method for a class that defines only one new method in its class part structure. This method is the Form widget's layout code.

Example 11-10. The class_part_init method of Form.

```
static void ClassPartInitialize(class)
    WidgetClass class;
{
    register FormWidgetClass c = (FormWidgetClass) class;

    if (c->form_class.layout == XtInheritLayout)
        c->form_class.layout = Layout;
}
```

The `XtInheritLayout` symbol is defined in the private include file for any class that defines new class part methods (one for each new method). Its value is always `_XtInherit`.

Form itself sets the `layout` field to a pointer to its `Layout` function. When its `class_part_init` method is called when the first instance of Form is created, it does nothing because the `layout` field is not `XtInheritLayout`. When a subclass is defined that sets the `layout` field to a function, the same thing happens: Form's `class_part_init` method is called because it is chained downward (the `class_part_init` methods of all superclasses are called), and it still does nothing because the `layout` field is not `XtInheritLayout`. Thus, the subclass has replaced Form's method. But if the subclass sets the `layout` field to XtInheritLayout, Form's `class_part_init` method sets the field to its own `Layout` function. The subclass has inherited Form's method.

Usually, only the class that defines a particular new method resolves the inheritance by checking for the value of that field in its `class_part_init` method. There is no point in a subclass also checking for an `XtInherit` value, since the downward chaining means that the superclass will have already processed and replaced the `XtInherit` value before the subclass `class_part_init` method is called.

11.3.6 The geometry_manager Method

`geometry_manager` methods handle requests from the children to be resized. Therefore, they typically use the proposed geometry passed in from the child to calculate a new experimental layout, and actually move and resize the children if the new layout is acceptable. However, when the request is just a query, the method should be able to return the same values without actually moving or resizing anything.

The Form `geometry_manager` method is shown in Example 11-11. Note that Form uses the `allow_resize` field (the `XtNresizable` resource) to determine whether to even consider the resize request. Then, if the request specifies a width and height, Form will accept the change by returning `XtGeometryYes`. The `XtMakeGeometryRequest` call that invoked the `geometry_manager` will actually make the geometry change before returning to the child's code. If the request specifies any other geometry change (border width, position, or stacking order), Form will deny the request. Finally, if the request was not a query, Form actually does the new layout. Note that Form never returns `XtGeometryDone` since it never makes the geometry changes itself. Instead it returns `XtGeometryYes` when it agrees with the changes, and lets Xt make the changes.

Note that the `allowed` structure in this routine could be replaced by individual *`width`* and *`height`* variables. Also note that the `reply` structure is never filled; it is used only when the `geometry_manager` method wants to suggest a compromise.

Example 11-11. Form: the geometry_manager method

```
/* ARGSUSED */
static XtGeometryResult GeometryManager(w, request, reply)
    Widget w;
    XtWidgetGeometry *request;
    XtWidgetGeometry *reply;    /* RETURN */
{
    FormConstraints form = (FormConstraints)w->core.constraints;
    XtWidgetGeometry allowed;

    if ((request->request_mode & ~(XtCWQueryOnly |
            CWWidth | CWHeight)) ||
            !form->form.allow_resize)
        return XtGeometryNo;

    if (request->request_mode & CWWidth)
        allowed.width = request->width;
    else
        allowed.width = w->core.width;

    if (request->request_mode & CWHeight)
        allowed.height = request->height;
    else
```

Example 11-11. Form: the geometry_manager method (continued)

```
        allowed.height = w->core.height;

    if (allowed.width == w->core.width && allowed.height ==
            w->core.height)
        return XtGeometryNo;

    if (!(request->request_mode & XtCWQueryOnly)) {
        /* reset virtual width and height. */
        form->form.virtual_width = w->core.width = allowed.width;
        form->form.virtual_height = w->core.height = allowed.height;
        RefigureLocations( (FormWidget)w->core.parent );
    }
    return XtGeometryYes;
}
```

The `RefigureLocations` called from the `geometry_manager` method is a private function analogous to the `DoLayout` routine used in ScrollBox, except that `RefigureLocations` calls Form's `layout` method that contains the actual layout code so that the method can be inherited or replaced by subclasses. The `layout` method calculates a layout and moves and resizes the children. `RefigureLocations` is also called from the `change_managed` method, as described in Section 11.4.9. Example 11-12 shows the `RefigureLocations` function and Form's `layout` method, which it calls. (The `if` statement that branches depending on the value of the `no_refigure` field allows an application to turn relayout on and off, as described in Section 11.4.11.)

Example 11-12. Form: private functions: RefigureLocations and the layout method

```
static void RefigureLocations(w)
    FormWidget w;
{
    /* no_refigure supports the relayout recalculation
      delay described later in this chapter  */
    if (w->form.no_refigure) {
        w->form.needs_relayout = True;
    }
    else {
        (*((FormWidgetClass)w->core.widget_class)->form_class.layout)
                ( w, w->core.width, w->core.height );
        w->form.needs_relayout = False;
    }
}

/* ARGSUSED */
static Boolean Layout(fw, width, height)
    FormWidget fw;
    Dimension width, height;
{
    int num_children = fw->composite.num_children;
    WidgetList children = fw->composite.children;
    Widget *childP;
    Position maxx, maxy;
    static void LayoutChild();
    Boolean ret_val;

    for (childP = children; childP - children < num_children;
            childP++) {
```

Example 11-12. Form: private functions: RefigureLocations and the layout method (continued)

```
        FormConstraints form = (FormConstraints)
                (*childP)->core.constraints;
        form->form.layout_state = LayoutPending;
    }

    maxx = maxy = 1;
    /*
     * Layout children one at a time, and determine
     * necessary size for self
     */
    for (childP = children; childP - children
            < num_children; childP++) {
        /*
         * Layout child then find position of bottom right
         * outside corner of child
         */
        if (XtIsManaged(*childP)) {
            Position x, y;
            LayoutChild(*childP);
            x = (*childP)->core.x + (*childP)->core.width
                    + ((*childP)->core.border_width << 1);
            y = (*childP)->core.y + (*childP)->core.height
                    + ((*childP)->core.border_width << 1);
            if (maxx < x) maxx = x;
            if (maxy < y) maxy = y;
    }
    }

    fw->form.preferred_width = (maxx += fw->form.default_spacing);
    fw->form.preferred_height = (maxy += fw->form.default_spacing);

    /* Now ask parent to resize us.  If it says Almost, accept the
     * compromise.  If Almost and parent chose smaller size, or No
     * and we were smaller than necessary, children will be clipped,
     * not laid out again.
     */
    if (fw->form.resize_in_layout
            && (maxx != fw->core.width || maxy != fw->core.height)) {
        XtGeometryResult result;
        result = XtMakeResizeRequest( fw, (Dimension)maxx,
                (Dimension)maxy, (Dimension*)&maxx, (Dimension*)&maxy );
        if (result == XtGeometryAlmost)
            result = XtMakeResizeRequest( fw, (Dimension)maxx,
                    (Dimension)maxy, NULL, NULL );
        fw->form.old_width  = fw->core.width;
        fw->form.old_height = fw->core.height;
        ret_val = (result == XtGeometryYes);
        } else ret_val = FALSE;

    return ret_val;
}
```

The `layout` method treats one child at a time, first initializing the `layout_state` private constraint instance field of each child to `LayoutPending`. The `LayoutChild` routine will start from this value. Next, it calls `LayoutChild` for each child, and at the same time keeps a running total of the sizes of the children so that when the loop is finished it

knows how big to be to fit all the children. Finally, it requests of its parent that it be just big enough to fit its children. If the parent denies the request, the code makes no attempt to make another request. If the parent offers a compromise, it is accepted. The Form widget, in either case, may be too big or too small to fit its children. If it is too small, some of its children will be clipped.

The `LayoutChild` routine is shown in Example 11-13. What it does is simple, although it is a little hard to follow because it is called recursively. It moves the child according to the `XtNfromHoriz` and `XtNfromVert` constraint resources.* These resources specify that a child be placed to the right of or below another particular child.

Example 11-13. Form: the LayoutChild private function

```
static void LayoutChild(w)
    Widget w;
{
    FormConstraints form = (FormConstraints)w->core.constraints;
    Position x, y;
    Widget ref;

    switch (form->form.layout_state) {

        case LayoutPending:
                form->form.layout_state = LayoutInProgress;
                break;

        case LayoutDone:
                return;

        case LayoutInProgress:
                String subs[2];
                Cardinal num_subs = 2;
                subs[0] = w->core.name;
                subs[1] = w->core.parent->core.name;
                XtAppWarningMsg(XtWidgetToApplicationContext(w),
                     "constraintLoop","xawFormLayout","XawToolkitError",
                     "constraint loop detected while laying out child\
                     '%s' in FormWidget '%s'",
                     subs, &num_subs);

                return;
    }

    x = form->form.dx;
    y = form->form.dy;

    if ((ref = form->form.horiz_base) != (Widget)NULL) {
        LayoutChild(ref);
        x += ref->core.x + ref->core.width +
                (ref->core.border_width
                << 1);
    }
    if ((ref = form->form.vert_base) != (Widget)NULL) {
        LayoutChild(ref);
        y += ref->core.y + ref->core.height +
                (ref->core.border_width
```

*Form resizes children only when it is resized—never during normal layout.

Example 11-13. Form: the LayoutChild private function (continued)

```
                << 1);
    }
    XtMoveWidget( w, x, y );
    form->form.layout_state = LayoutDone;
}
```

If neither `XtNfromHoriz` nor `XtNfromVert` are set for the child, it is simply placed the default distance from the top-left corner of the Form. When one child is set, the next child must be placed relative to that child. However, the other child may be later in the list and not properly positioned yet. Therefore, the code calls `LayoutChild` to lay out the child that this child is positioned relative to.

The `layout_state` field catches circular settings for the `XtNfromHoriz` and `XtNfromVert` resources. For example, if widget *A* is specified to the right of widget *B*, and widget *B* is specified to the right of widget *A*, there is no solution. `LayoutChild` would be caught in an infinite loop of calling itself. When first called from the `layout` method, the `layout_state` is `LayoutPending`. This is changed to `LayoutInProgress` in the `switch` statement. If the function is called again for the same child, this state will cause the warning message to be printed and the function to exit. The Form widget does not exit—it just gives up processing the invalid constraint resource setting and prints a warning message.

11.3.7 The resize Method

The `resize` method calculates a layout to fit in the new dimensions of Form and moves and resizes the children accordingly. Form's `resize` method is shown in Example 11-14. It consists of a loop that treats each managed child one at a time. The position and dimensions of each child are calculated with the help of the private function `TransformCoord` (also shown in Example 11-14) and the child is moved and resized. `TransformCoord` handles one parameter at a time, and uses a position, the size before resizing, the size after resizing, and the constraints settings to arrive at the appropriate value for the parameter. The old width and height of the Form widget are initialized in the Core `initialize` method and updated at the end of the `resize` method.

Example 11-14. Form: the resize method

```
static void Resize(w)
    Widget w;
{
    FormWidget fw = (FormWidget)w;
    WidgetList children = fw->composite.children;
    int num_children = fw->composite.num_children;
    Widget *childP;
    Position x, y;
    Dimension width, height;

    for (childP = children; childP - children < num_children;
            childP++) {
        FormConstraints form = (FormConstraints)
                (*childP)->core.constraints;
        if (!XtIsManaged(*childP)) continue;
```

Example 11-14. Form: the resize method (continued)

```
        x = TransformCoord( (*childP)->core.x, fw->form.old_width,
                fw->core.width, form->form.left );
        y = TransformCoord( (*childP)->core.y, fw->form.old_height,
                fw->core.height, form->form.top );

        form->form.virtual_width =
                TransformCoord((Position)((*childP)->core.x
                + form->form.virtual_width
                + 2 * (*childP)->core.border_width),
                fw->form.old_width, fw->core.width,
                form->form.right )
                - (x + 2 * (*childP)->core.border_width);

        form->form.virtual_height =
                TransformCoord((Position)((*childP)->core.y
                + form->form.virtual_height
                + 2 * (*childP)->core.border_width),
                fw->form.old_height, fw->core.height,
                form->form.bottom )
                - ( y + 2 * (*childP)->core.border_width);

        width = (Dimension)
                (form->form.virtual_width < 1) ? 1 :
                form->form.virtual_width;
        height = (Dimension)
                (form->form.virtual_height < 1) ? 1 :
                form->form.virtual_height;

        XtConfigureWidget( *childP, x, y, (Dimension)width,
                (Dimension)height, (*childP)->core.border_width);

    }

    fw->form.old_width = fw->core.width;
    fw->form.old_height = fw->core.height;
}

static Position TransformCoord(loc, old, new, type)
    register Position loc;
    Dimension old, new;
    XtEdgeType type;
{
    if (type == XtRubber) {
        if ( ((int) old) > 0)
            loc = (loc * new) / old;
    }
    else if (type == XtChainBottom || type == XtChainRight)
        loc += (Position)new - (Position)old;

    return (loc);
}
```

This `resize` method stores the new size of the children in the `virtual_width` and `virtual_height` constraint part fields, and uses their previous values to arrive at the new size. This is done because Form's `XtNtop`, `XtNbottom`, `XtNleft`, and `XtNright` constraints specify the geometry of the child based on its previous geometry.

Notice that the `for` loop in this particular `resize` method loops through the children directly, using pointer arithmetic. This is equivalent to using a loop that increments an integer and then uses the integer to index the `children` array. For example, the first five lines of the loop could also be expressed as:

```
int i;

for (i = 0; i < num_children; i++) {
    FormConstraints form = (FormConstraints)
            (children[i])->core.constraints;
    if (!XtIsManaged(children[i])) continue;
    x = TransformCoord( (children[i])->core.x,
            fw->form.old_width, fw->core.width, form->form.left );
        .
        .
        .
}
```


11.3.8 The Core and Constraint set_values Methods

When the application calls `XtSetValues` to set the resources of a child of a constraint widget, Xt calls the child's Core `set_values` method and then the parent's Constraint `set_values` method. Both methods are passed the same arguments. Constraint `set_values` validates the ranges of constraint resource settings and computes the value of any private constraint instance part fields that depend on constraint resource values. It should also set child Core geometry fields to match the changes in constraint resources. For example, if a constraint for the maximum height of a widget is changed to a value smaller than the widget's current height, then the Constraint `set_values` procedure should reset the height field in the widget.

Both Core and Constraint `set_values` must return `TRUE` or `FALSE` to indicate whether redisplay of the widget is necessary. For composite and constraint widgets, this value is usually meaningless because there is nothing to redisplay. But these might be useful if, for some reason, you write a composite widget that does have display semantics.

Form defines both the Core and Constraint `set_values` methods as empty functions that return `FALSE`. An easier way to do this is to specify `NULL` for them in the class structure initialization.

11.3.9 The change_managed Method

The `change_managed` method is responsible for making the initial layout of an application and changing the layout when any child changes management state. Form's `change_managed` method (shown in Example 11-15) calls `RefigureLocations` to actually do a layout. (`RefigureLocations` is a private routine equivalent to `DoLayout` in ScrollBox, described in Section 11.4.6.) Form's `change_managed` method also stores the previous size of the children in the `virtual_width` and `virtual_height` constraint part fields for use in the `resize` method as described in Section 11.4.7.

Example 11-15. Form: the change_managed method

```
static void ChangeManaged(w)
    Widget w;
{
    FormWidget fw = (FormWidget)w;
    FormConstraints form;
    WidgetList children, childP;
    int num_children = fw->composite.num_children;
    Widget child;

    /*
     * Reset virtual width and height for all children.
     */

    for (children = childP = fw->composite.children;
             childP - children < num_children; childP++) {
        child = *childP;
        if (XtIsManaged(child)) {
            form = (FormConstraints)child->core.constraints;

            if ( child->core.width != 1)
                form->form.virtual_width = (int) child->core.width;
            if ( child->core.height != 1)
                form->form.virtual_height = (int) child->core.height;
        }
    }
    RefigureLocations( (FormWidget)w );
}
```

11.3.10 The query_geometry Method

Form's `query_geometry` method (shown in Example 11-16) is the minimal version almost identical to the one described for simple widgets in Chapter 6, *Basic Widget Methods*. The `preferred_width` and `preferred_height` instance variables are set in the Form class `Layout` method to the size that just fits the current layout.

Example 11-16. Form: the query_geometry method

```
static XtGeometryResult PreferredGeometry( widget, request, reply  )
    Widget widget;
    XtWidgetGeometry *request, *reply;
{
    FormWidget w = (FormWidget)widget;

    reply->width = w->form.preferred_width;
    reply->height = w->form.preferred_height;
    reply->request_mode = CWWidth | CWHeight;
    if (  request->request_mode & (CWWidth | CWHeight) ==
            reply->request_mode & CWWidth | CWHeight
            && request->width == reply->width
            && request->height == reply->height)
        return XtGeometryYes;
    else if (reply->width == w->core.width && reply->height ==
            w->core.height)
        return XtGeometryNo;
```

Example 11-16. Form: the query_geometry method (continued)

```
    else
        return XtGeometryAlmost;
}
```

11.3.11 Delaying Geometry Recalculation

During an application's initial layout, the `change_managed` method of a composite widget is called only once even though many children may have been managed. However, after that, `change_managed` is called once for every child that changes management state. Many composite or constraint widgets, especially ones that have complicated layout code, provide a public function (such as the one shown in Example 11-17) that the application can call to turn off layout recalculation until a group of windows is managed or unmanaged, and then call again to trigger recalculation once the whole group of children has been managed or unmanaged.

To implement this delay, you need an instance variable to hold a Boolean value indicating whether to delay or not (`no_refigure`, in this case). You set and unset this variable in this public routine and you test it in `change_managed`.

Example 11-17. Form: the public function for delaying calls to change_managed

```
void XawFormDoLayout(w, doit)
Widget w;
Boolean doit;   /* FALSE, don't recalculate; TRUE, do */
{
    register FormWidget fw = (FormWidget)w;

    fw->form.no_refigure = !doit;

    if ( XtIsRealized(w) && fw->form.needs_relayout )
        RefigureLocations( fw );
}
```

11.4 Compound Widgets

A compound widget is a combination of widgets that are put together to make a higher-level, user-interface object. For example, Xaw makes the Dialog widget by combining the Label and Text or Command widgets in a widget that is actually a trivial subclass of the Form widget. Dialog creates the Label, Text, and Form widgets in its `initialize` method, and sets constraint resources to position them. Dialog provides its own resource list to allow an application to configure some characteristics of its children. The application can manipulate the children only through these resources, because it cannot access the widget IDs of the subwidgets of Dialog without breaking the rules of data hiding. Thus, compound widgets provide programming convenience, but they make it more difficult to take advantage of all the configurable aspects of the subwidgets.

The main widget of the compound widget may be a subclass of Core, Composite, or Constraint. If it is a subclass of Core, the widget manages the positions and sizes of its children manually whenever it is resized. The success of this strategy is dependent on the children never trying to resize themselves and on the application never trying to resize the children directly.* The latter will not be a problem unless the application breaks the data-hiding rules by manipulating the child directly. The Text widget is an example of this kind of widget. It creates and manages its own scrollbar.

Compound widgets normally define only a few methods and inherit the rest. Compound widgets based on Core will move and resize their children manually in their `resize` method. If the widget is a subclass of Composite or Constraint, the normal geometry management facilities manage the position and size of the children. If it is a subclass of Constraint, the main widget sets the constraints of the children to control the geometry management process by providing a Constraint `initialize` method.

A compound widget always needs a `destroy` method that destroys the children it created.

Compound widgets also need a `set_values` method to manage their resources.

11.5 Stacking Order

We promised earlier to say a bit more about how composite or constraint widgets can control the stacking order of their children. We noted that this must be done manually, because Xt doesn't provide much support for it. This is because most applications do not stack widgets—the whole concept of geometry management is based on each widget trying to lay out its children *without* stacking them. However, there are applications where it makes sense to stack widgets. For example, an application that provides note cards, where each card is a widget, would want to stack them showing only the corner of hidden cards.

There is no Core resource for stacking order, and therefore it can't be set with `XtSetValues` unless you define the resources in your own widget class. Xt provides no call to restack windows; you must use the Xlib functions `XConfigureWindow`, `XRestackWindows`, `XRaiseWindow`, or `XLowerWindow`. When a widget suggests a stacking order for itself through its `query_geometry` method, Xt takes care of making the required Xlib call if the parent agrees with the change. However, stacking requests of unrealized widgets have no effect (so stacking order won't be set this way in the initial geometry negotiation). Therefore, the most robust method to handle stacking order is for your composite widget to make the appropriate Xlib calls directly to change the stacking order of its children. `XRestackWindows` is probably the best call to use. Since restacking the windows doesn't change their requirements for screen space, it shouldn't affect either the parent or the children adversely. The appropriate place to call `XRestackWindows` depends on when you want to change the stacking order. (Note that the stacking change won't become visible until the next time Xt is waiting for an event.)

*When a child of a simple widget calls `XtMakeGeometryRequest` because it wants to change its size, `XtMakeGeometryRequest` always makes the requested changes and returns `XtGeometryYes`. Therefore, a simple widget parent really has no control over its child if the child wants to resize itself. A simple widget cannot even tell that the child has resized itself.

You can control the initial stacking order of a group of children by creating them in the desired order. The most recently created widget appears on the bottom. (This is the opposite of what you might expect if you know that newly created X windows appears on top of their siblings. The difference is due to the way a composite widget maintains its list of children.)

12

Menus, Gadgets, and Cascaded Pop Ups

This chapter describes how menus work, and several ways to create menu widgets. One of these ways involves the use of windowless widgets, or gadgets. This chapter also describes how to use more advanced features of the Xt pop up mechanism, including modal cascades, to implement cascading pop up menus.

In This Chapter:

Menu Styles and Implementation 359
- How Menus are Popped Up 360
- Menu Panes 361

Several Ways to Create and Use Menus 363
- A Spring-Loaded Menu: Pointer Grabbing 363
- A Drop-Down Menu 371
- Cascaded Menus 374
- Using the R4 SimpleMenu Widget 379
- Delayed Popup Creation 383

About Dialog Boxes 384

Gadgets 385
- Inside a Gadget 388
- Private Header File 389
- The Gadget Source File 390
- The Public Header File 391
- The Gadget Parent 391

12
Menus, Gadgets, and Cascaded Pop Ups

In Chapter 2, *Introduction to the X Toolkit*, we show a simple example that pops up a dialog box. This chapter is much more thorough in describing the Xt facilities for managing pop ups, including both pop-up menus and dialog boxes.

Although the menus provided by various widget sets vary greatly in the way they look and in the way they are used in the application, the underlying Xt facilities for managing them are the same. This chapter presents a series of examples based on R4 Athena widgets that implement menus in different ways. While some of the techniques shown may be hidden within the menu widgets provided in commercial widget sets, it will help you understand menus better to see the underlying techniques fully exposed and explored.

This chapter also discusses cascaded pop ups—pop ups that call other pop ups—and the event management necessary to have pop ups that shut out other input elsewhere in the application and system.

Finally, this chapter discusses windowless widgets called *gadgets*, which have been designed to reduce window system overhead. Their most important use is to implement the panes in menu widgets. Gadgets were originally developed by Digital as a part of DECWindows, and were carried over into the version of Xt shipped with Motif 1.0. The MIT X Consortium Intrinsics support gadgets beginning in Release 4 (R4). As mentioned earlier, Motif 1.0 supplies its own version of the Intrinsics that is not compatible with the MIT X Consortium Intrinsics. However, Motif 1.1 adopts the standard R4 Intrinsics. As an example of a widget that manages gadgets, we will show the R4 Athena SimpleMenu widget and its gadget children.

Also introduced in R4 is the *object*, which is another kind of windowless widget even simpler than a gadget. Objects are not usually used in menus, so we will reserve discussion of them until Chapter 13, *Miscellaneous Toolkit Programming Techniques*.

In this chapter, we are using the term *menu* broadly, to refer to any user-interface element that lists many options and allows the user to select one or more. A menu might consist of a list of commands, only one of which can be selected at a time, a list of nonexclusive Boolean settings that can be turned on or off, or a list of exclusive choices (such as the colors or patterns for a paint palette). A menu that invokes commands will start in the same state each

time, while the other two types may have different contents in any particular invocation, showing the settings invoked the previous time or all previous times, or a modified list of choices.

Menus are one of the most important user-interface elements in window-based applications. They offer the same feature as Command buttons—a way for the user to invoke application functions or set parameters—but in a more organized and more easily accessible fashion when there are more than a few buttons.

Figure 12-1 compares a menu to a box full of buttons.

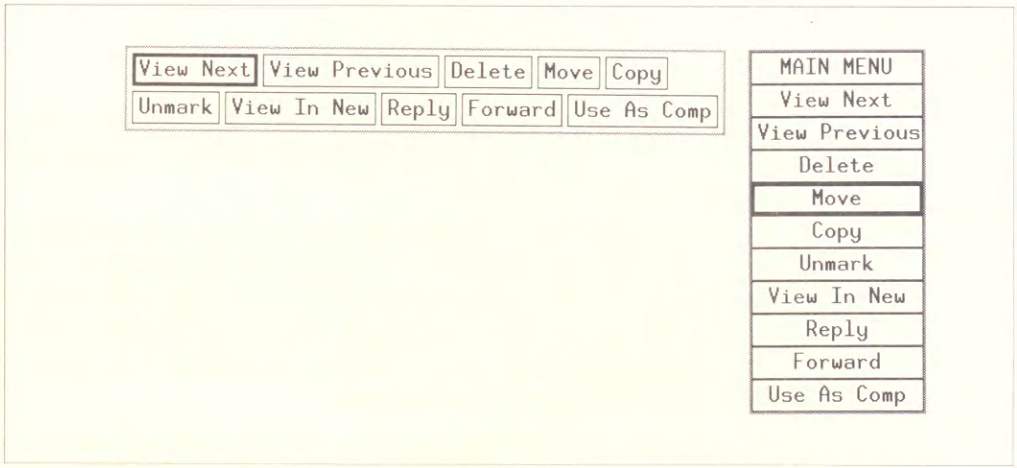


Figure 12-1. Command widgets in a button box, and the same commands as a menu

The menu takes up less space because only its title is visible until it is called up.* As a result, you can have more menus than you could have permanent button boxes. Commands can be presented in smaller, more closely related groups. The user will spend less time searching for the desired command.

The commands in the menu are also easier to read because they are arranged one per row. The commands in the menu may even be easier to invoke because it is more natural to drag the mouse up and down than from side to side. And last but not least, menus avoid the worst problem with button boxes: when the application is resized, button boxes may place each command widget in a different position, making it more difficult for the user to find commands.†

*Under some widget sets and window managers, menus don't even display a title—they simply pop up at the pointer position in response to a particular pointer button/keypress combination. This is the behavior of the menus provided by *xterm* and *uwm*. However, this is not very desirable behavior from a user-interface point of view, since it gives the user no visual feedback that a menu is available or how to invoke it. The user needs the manual—something graphical user interfaces are designed to avoid.

†To be fair, there is something to be said for the fact that all the available commands are always visible in an application that uses button boxes. You can invoke a button in a box with just a button click, while in a menu it requires a press, a drag, and a release. When there are only a small number of commands, putting the command widgets in a box is probably better than using a menu.

Some of the applications in the core distribution from MIT use button boxes instead of menus because there was no menu widget in the Athena widget set until Release 4.* And some of the applications that do use menus have implemented them directly with Xlib.

12.1 Menu Styles and Implementation

The conventions for the appearance and user interface of menus (look and feel) in widget sets probably varies more than any other aspect of the user interface.

There are several different styles of menus. As we've pointed out earlier, a button box is itself a style of menu. However, in this chapter, we will be focusing on pop-up menus—menus that are not visible until the user presses a pointer button or a key-button combination.

There are several different styles of pop-up menu. Probably the most familiar is the pull-down menu popularized by the Apple Macintosh. A *pull-down menu* has a label permanently visible in the application, usually on a menu bar at the top. When the pointer is clicked on the label, and then dragged downwards, the menu is pulled down like a window shade, and remains displayed as long as the pointer button is depressed. The currently selected item (as indicated by the pointer position within the menu) is highlighted, and is executed when the pointer button is released.

The variation adopted by both Motif and OPEN LOOK (possibly to avoid legal entanglements with Apple) is the *drop-down menu*. The pointer need not be dragged down to display the menu. Instead, it appears below the menu title as soon as the button is depressed in the menu title. The distinction between pull-down and drop-down menus is a subtle one, since with a drop-down menu the pointer must subsequently be dragged down the menu in order to make a selection.

In some cases, selecting an item on the menu or moving off the right side of certain menu panes causes a second menu to appear next to the first (usually to the right). This is referred to as a *cascading pop up*. (Another type of cascading pop up is a dialog box that pops up another dialog box.)

Finally, there is the pure *spring-loaded pop-up menu* used by many of the standard X clients, which displays no menu label, and simply pops up at the pointer position, given the appropriate key or button press. For example, the menus in *xterm* pop up when you hold the Control key and press the first or second button while the pointer is anywhere in the *xterm* window.

One can also imagine many other possible menu styles. For example, an effective user interface could be constructed using only pop-up button boxes, emulating the single-line menu popularized by Lotus for its character-based 123 spreadsheet. Any given button might either execute an action, or pop up a lower-level menu, which would overlay (and thus appear to replace) the first menu.

*The R4 menu widgets will not work under R3 even if you can get the code for them, because they depend on changes to Xt made in R4.

In this chapter, though, we will focus on the two styles of menu you are most likely to encounter in X applications: the drop-down menu and the pure spring-loaded pop-up menu.

The difference between spring-loaded and drop-down menus is primarily in the method by which the user invokes the menu and in where the menu is placed; one menu widget class can usually work in either style.

The Xt specification makes the distinction between modeless pop ups, modal pop ups, and spring-loaded pop ups.

Modeless pop ups are windows that, once popped up, are subject to window manager control, and for all intents and purposes act like regular applications in themselves. A help window that stayed up, and could be moved and resized like a regular window once popped up, is an example of this type of pop up. It is referred to as "modeless" because it doesn't put the application into a special mode, in which only input to the pop up is allowed.

A modal pop up may or may not be visible to the window manager, but it always disables user-event processing by the application, except in the pop up itself. A dialog box that requires the user to enter data or click on a button is an example of a modal pop up. Input may still be possible to other applications.

A spring-loaded pop up, as defined by Xt, is invisible to the window manager, and disables user input to all windows in all applications, except to the pop up itself. The most important thing about spring-loaded pop ups is that they are invoked with a key or pointer button press, whereas another type of pop up might be invoked as a routine part of application processing, or just because the pointer entered a particular window.

However, due to a lack of appropriate terminology, throughout this chapter we use the term "spring-loaded pop up" to refer to menus that pop up at the pointer position when a mouse button is pressed, such as that used by *xterm*, as opposed to drop-down or pull-down menus.

12.1.1 How Menus are Popped Up

How you create menus in an application differs according to the class of menu widget and whether it will be drop-down or spring-loaded. Commercial widget sets are designed to make it quite easy to create menus that fit into their user-interface conventions. As usual, the examples in this chapter use the Athena widgets to implement various types of menus. Although the procedure for creating menus under the widget set you plan to use may be different, many of the underlying issues are the same.

In Section 3.3, a dialog pop up was created by first creating a pop-up shell, and then creating the widget to be popped up as a child of the pop-up shell. This procedure is used for some menu widgets, but most menu widgets are themselves subclasses of Shell, and therefore no separate shell needs to be created. You just create an instance of the menu widget itself, using `XtVaCreatePopupShell` instead of `XtVaCreateManagedWidget`.

To have a spring-loaded pop up, your application usually adds an action that places the widget in the application main window. (Xt has a standard action for popping up a widget, but by default it places the widget at the top-left corner of the screen. As a result, you must

use it in conjunction with an action of your own, which moves the invisible shell widget to the desired location before it is actually popped up.) The application-defaults file should supply a translation to specify what event sequence will call up that menu.

In some cases, a menu widget itself may add actions and translations to the parent so that the widget can be popped up in spring-loaded fashion without any application code. In others, the widget will add the actions but leave the user or the application writer to define the translation that will invoke the action.

To use a menu in drop-down fashion, you create a Command widget and use its callback to pop up the menu, or better still, use a widget specially made to invoke a drop-down menu, and tell it the name of the shell to be popped up (which as just mentioned may be the menu widget itself).

Menus differ widely on their conventions for when to pop down. Most menus pop down after a choice is made. In this case, the callback invoked to tell the application about the menu choice would pop down the widget, or perhaps the menu widget itself would pop itself down before calling the chosen item's callback. Some menus pop down whenever the button that popped them up is released (for example, *xterm*), even if the pointer is outside the menu when the button is released. Some pop down whenever the pointer moves out of the menu (for example, *uwm*).* OPEN LOOK menus provide a pushpin that allows the user to keep a pop-up menu on the screen, where it will be handy for repeated use. The pushpin looks like a thumb tack and has two positions: fastened and loose.

12.1.2 Menu Panes

The style of the items, or *panes*, in the menus also varies widely. Panes are usually implemented using one or more subwidgets, or in the case of Motif and the Athena SimpleMenu widget, with gadgets, which are windowless objects that can be drawn and managed separately within a widget. There may be several different widgets or gadgets that implement menu panes with different input and output semantics.

If the panes are widgets, they are created with `XtVaCreateManagedWidget`, or in some menu widgets, by specifying resources of the menu widget, which will then create its own panes; how gadgets are created is discussed later in this chapter.

Figure 12-2 shows menus developed using OSF's Motif and AT&T's OPEN LOOK Toolkit, which you can compare and contrast with each other and with the R4 Athena SimpleMenu widget shown in Figure 12-1.

*Most people find it annoying to have menus that pop down when the pointer moves out of them. It is too easy to accidentally move outside the menu and have to start all over. All the menus in this chapter pop down the menu when the choice is made, or when the button is released outside the menu.

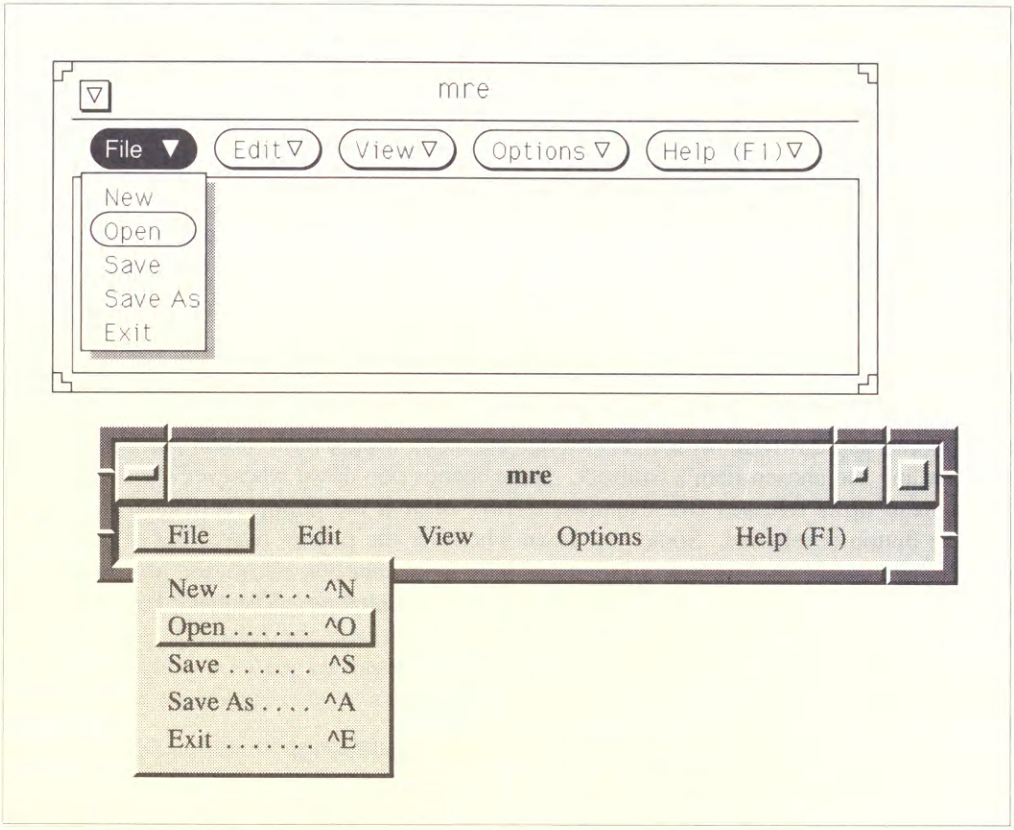


Figure 12-2. Menus from the OPEN LOOK and Motif widget sets

The Athena SimpleMenu widget has rectangular panes that display an identifying string, and an optional bitmap before and/or after the string. For example, the bitmap can be used to place a checkmark to the left of the label (indicating that an option has been selected and is in effect), or an arrow to the right, indicating that selecting this pane will produce a cascading submenu. When an item is selected, it is displayed in inverse video.

Motif menu panes are rectangular. The currently-selected pane is highlighted with a simulated 3-D shadow. Panes may contain either a string or a bitmap.

OPEN LOOK menus come with several types of panes, depending on the type of menu item. Commands, whether available from menus, or in other control areas, are represented by oblong buttons with rounded ends. (Buttons that generate menus or submenus follow their label with an arrowhead pointing in the direction where the submenu will appear.) Exclusive options are shown as adjacent rectangles, with the one currently chosen highlighted by a dark border. Nonexclusive options are displayed in rectangles with a small separation between each one. Panes may contain a string or a bitmap, or both.

However, what all menu widgets have in common is that the application programmer can set the label of each pane and the function it invokes. Exactly how to do this varies according to the widget set you use.

See Appendix A, *OPEN LOOK and Motif*, for more information on how Motif and OPEN LOOK implement menus.

12.2 Several Ways to Create and Use Menus

You already know how to create a pop-up shell, create a Box as its child, fill it with Command widgets, label the Command widgets, and register a callback function for each one. By doing this you can create a basic menu. Its performance may be rather sluggish since it unnecessarily consumes resources, but it looks and works like a menu.

This section describes how to use the menu made from a box full of buttons in two different menu styles: spring-loaded and drop-down. The purpose of this exercise is to expose some of the issues involved in event-management of pop ups. Some of these issues may be hidden in the widgets or facilities provided by some commercial widget sets, but seeing how to do the event management explicitly should help you to use pop ups more effectively.

The challenge of creating a pop up with Box and Command buttons is to make it pop up and down at the right times, and to control its event handling to fit the menu style. We will also experiment with creating a cascaded menu, in which one menu pane in a main menu invokes a submenu.

Finally, this section describes how to create a menu using the R4 Athena SimpleMenu widget and its gadget children.

12.2.1 A Spring-Loaded Menu: Pointer Grabbing

A spring-loaded menu should pop up when a button press occurs in a particular widget; usually the application's main window. The menu should stay visible as long as the user holds down that button, and disappear when the button is released. If the button is released in a menu pane, the function registered for that pane should be invoked. If the button is released outside the menu, no function should be invoked but the menu should still be popped down.

The only tricky part of implementing a spring-loaded menu is getting the menu to pop down when the button is released outside the menu. Since this occurs outside the menu and possibly outside the application, the X server will not send the button release event to the application unless a *grab* is in effect. Normally, user events are sent to the window that contains the pointer. But after an application makes a grab, the X server sends all events of particular types to the window that made the grab, even if the pointer is no longer in the window.

The X server defines several types of grab: keyboard grabs, pointer grabs, and server grabs. Keyboard and pointer grabs control only input from the indicated device, while server grabs make the server act on requests from one application exclusively. (Server grabs are mainly used by window managers.) Pointer grabs are used for controlling events in pop ups when a pointer button pops up the pop up, and keyboard grabs are used when a key press pops up the pop up. We will discuss pointer grabs since keyboard grabs are analogous (and keyboard-triggered pop ups are less common).

There are two types of pointer grabs: passive grabs and active grabs. An *active grab* is invoked directly with the Xt function `XtGrabPointer`. This function tells the server that you want the grab to begin right away and to continue until specifically released with `XtUngrabPointer`. Active grabs are not normally used for pop ups.*

A *passive grab* tells the server that you want a grab to begin when a certain key or button combination is pressed in a certain window (the combination that is to pop up the pop up). The grab continues until the button in the combination is released. This is perfect for menus because we need the grab only until the button is released. (Also, as you'll see in the section on drop-down menus, you can register several passive grabs for the same key-button combination as long as each grab is initiated by a press in a different window. This technique lets you have as many drop-down menus as you want. Since spring-loaded pop ups are generally invoked by a press in the same window—the application main window—you will need to use a different key-button combination for each different menu.)

Passive grabs of a key or button and active grabs of the pointer or the keyboard we will call *global grabs*, since they affect not only this application but prevent distribution of the grabbed events to other applications running on the same server. This terminology is to distinguish the global grab from the effects of Xt's local *grab mode*, which simulates a global grab but requires no call to the server and affects only the distribution of events within the application. The Xt grab mode cannot commandeer events that occur outside the application like a global grab can. When an Xt grab is in effect, Xt redirects user events to the pop ups even if they occur somewhere else in the application. (Non-user events continue to be dispatched to widgets so that they can redraw themselves).

The Xt grab mode can be either exclusive or nonexclusive. Exclusive and nonexclusive Xt grabs differ only when a pop up has popped up another pop up—a so-called cascaded pop up. An *exclusive* Xt grab redirects all user events that occur within the application to the latest pop up in the cascade. A nonexclusive Xt grab redirects events to whichever pop up the pointer is in, or the latest pop up if the pointer is outside all the pop ups (but still in the application).

Here are examples of the two kinds of Xt grab modes. Consider an application that pops up a dialog box to get a filename from the user. The application wants to read the file. If the file can't be opened, the application pops up another dialog telling this to the user. This error pop up takes no input, so input is still desired in the filename entry pop up. This situation calls for a nonexclusive grab. By contrast, consider an application that uses the same filename entry pop up to save a file. If the file exists, it would pop up a dialog that would ask whether the existing file should be overwritten. This pop up must be answered before a new filename is chosen. This situation would call for an exclusive grab. In brief, an exclusive grab constrains input to the the latest widget in the cascade, while a nonexclusive grab allows input to any widget in the cascade. We'll talk more about the `Xtgrab` mode in Section 12.2.3, when we talk about pop-up cascades.

*One reason that `XtGrabPointer` is rarely used for pop ups is that it requires that the window that will receive the grabbed events be visible. This is often not the case. In a menu, for example, the window that you want to grab the events may be hidden by the menu panes even when the menu is popped up. Another reason is that you need to call `XtUngrabPointer` to release the grab when finished. Passive grabs match the task better.

Xt provides three ways of popping widgets up and down:

- There are three built-in callback functions: `XtCallbackNone`, `XtCallbackExclusive`, and `XtCallbackNonexclusive`. Each of these functions pops up a widget with a different type of Xt grab mode, as indicated by its name. `XtCallbackNone` makes no grab at all. `XtCallbackExclusive` makes an exclusive Xt grab, while `XtCallbackNonexclusive` makes a nonexclusive grab.

  `XtCallbackPopdown` is the corresponding built-in callback function to pop down a widget.

- There are two built-in actions, `XtMenuPopup` and `XtMenuPopdown`, that pop up or pop down a widget. You can use these actions in translation tables in the application-defaults file or application code. `XtMenuPopup` always asserts an exclusive Xt grab and a passive global grab if the pointer or keyboard invoked it.

- There are three functions, `XtPopup`, `XtPopupSpringLoaded`, and `XtPopdown`, that you can call directly in your application code to pop up or pop down a widget. `XtPopup` has a *`grab_kind`* argument that lets you specify whether to assert an exclusive or nonexclusive grab mode, or no grab. `XtPopupSpringLoaded` always asserts an exclusive Xt grab and makes an global pointer grab so that a button release outside a menu can be used to trigger the popping down of the menu.

Each of these ways is appropriate for different situations, and they are often used in combination. Each will be described in the sections below.

Passive global grabs can be invoked directly using the Xt functions `XtGrabButton` and `XtGrabKey`. However, Xt takes care of making the appropriate passive global grab if you use `XtPopupSpringLoaded` or the `XtMenuPopup` action to pop up your shell widget. As we will see, the other ways to pop up a widget do not make any global grab, which makes them inappropriate for popping up main menus.* However, they are still useful for some types of dialog boxes and for cascading submenus, if the main menu has used `XtMenuPopup` to assert an exclusive grab. We'll return to this subject in Section 12.2.3.

A pop-up menu globally grabs the pointer to force the user to make a menu choice before leaving the menu, or to pop down the menu if no choice is made. However, this global grab is necessary for another reason as well. The X server automatically grabs the pointer beginning at a button press and ending at the release of the same button. Since the initial button press pops up the menu, the next button release would also arrive at the same widget—the application main window or the menu title—even if the pointer were already in the menu. Furthermore, the application main window would get all `EnterNotify` and `LeaveNotify` events, so the Command widgets in the menu wouldn't get any of them. When the menu is popped up with `XtMenuPopup` action, the passive global grab it makes cancels the automatic global grab. (Again, pointer grabs redirect only the events caused directly by the pointer; `ButtonPress`, `ButtonRelease`, `EnterNotify`, `LeaveNotify`, and

*With `XtPopupSpringLoaded` you can write callback functions that are the equivalent of `XtCallback*`, except that they will pop up a menu with the necessary passive global grab. You can also write your own version of the `XtMenuPopup` action if it doesn't do what you want. You will need `XtRegisterGrabAction` to do this. This function tells Xt to automatically start a passive global grab whenever a certain action is invoked.

`MotionNotify`. All other events (most notably `Expose` events) occur and are delivered normally.)

With that background, let's take a look at an application that provides a spring-loaded menu. The *xmenu1* application's permanent appearance is a variation of *xbox*; it displays a large Label widget that we are using to simulate an application's main window and a Command widget for quitting. Pressing any button in the Label widget calls up the menu, which operates as described at the beginning of this section. *xmenu1* is shown in Figure 12-3. As usual, we suggest you compile and run this example now.

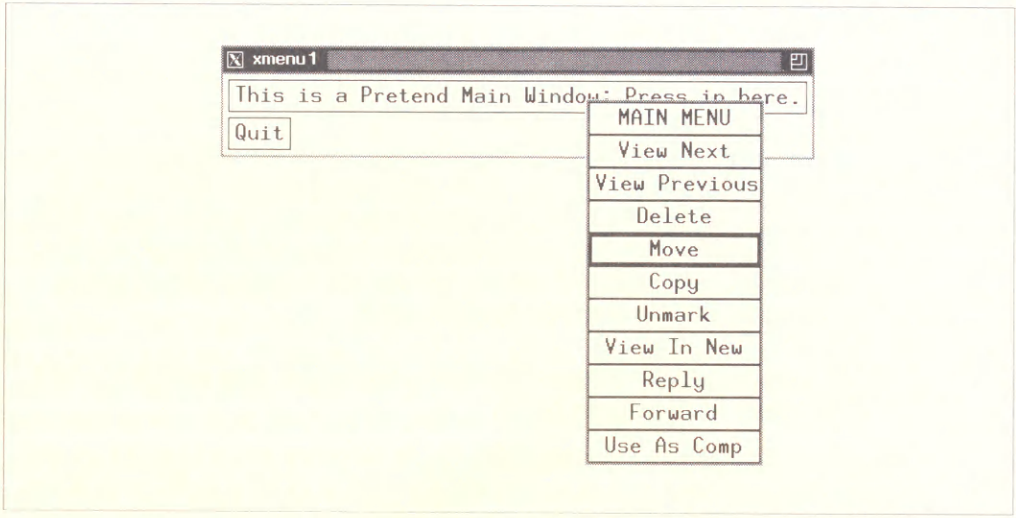


Figure 12-3. xmenu1: application with spring-loaded pop-up menu

The relevant code in *xmenu1* consists of an action routine to place the pop up, code to add the action, a callback routine to handle when a menu item has been chosen, and code to create the Box populated with Command widgets that will act as the menu. Example 12-1 shows the complete code.

Example 12-1. xmenu1: complete code

```
/*
 *  xmenu1.c - simple spring-loaded menu
 */

#include <stdio.h>

/*
 * Standard Toolkit include files:
 */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Shell.h>

/*
 * Public include files for widgets used in this file.
 */
```

Example 12-1. xmenu1: complete code (continued)

```
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Label.h>

/*
 * The popup shell ID is global because both dialog and pshell
 * are needed in the dialogDone callback, and both can't be
 * passed in without creating a structure.
 */
Widget pshell;

/*ARGSUSED*/
void PlaceMenu(w, event)
Widget w;
XButtonEvent *event;
{
    /* should make sure coordinates allow menu to fit on screen */

    /* move submenu shell to slightly left and above button
     * press position */
    XtVaSetValues(pshell,
            XtNx, event->x_root - 10,
            XtNy, event->y_root - 10,
            NULL);
}

/*
 * quit button callback function
 */
/*ARGSUSED*/
void Quit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
    exit(0);
}

/*
 * menu pane button callback function
 */
/*ARGSUSED*/
void PaneChosen(w, client_data, call_data)
Widget w;
XtPointer client_data;   /* cast to pane_number */
XtPointer call_data;
{
    int pane_number = (int) client_data;
    printf("Pane %d chosen.\n", pane_number);
    XtPopdown(pshell);
}

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_context;
    Widget topLevel, box, quit, label, menulabel, menubox, menupane[10];
    int i;
```

Example 12-1. xmenu1: complete code (continued)

```
String buf[50];

static XtActionsRec trial_actions[] = {
    {"placeMenu", PlaceMenu},
};

topLevel = XtVaAppInitialize(
    &app_context,       /* Application context */
    "XMenu1",    /* application class name */
    NULL, 0,            /* command line option list */
    &argc, argv,        /* command line args */
    NULL,               /* for missing app-defaults file */
    NULL);              /* terminate varargs list */

box = XtCreateManagedWidget(
    "box",              /* widget name */
    boxWidgetClass,     /* widget class */
    topLevel,           /* parent widget*/
    NULL,               /* argument list*/
    0                   /* arglist size */
    );

label = XtCreateManagedWidget(
    "label",            /* widget name */
    labelWidgetClass,   /* widget class */
    box,                /* parent widget*/
    NULL,               /* argument list*/
    0                   /* arglist size */
    );

quit = XtCreateManagedWidget(
    "quit",             /* widget name */
    commandWidgetClass, /* widget class */
    box,                /* parent widget*/
    NULL,               /* argument list*/
    0                   /* arglist size */
    );

pshell = XtCreatePopupShell(
    "pshell",
    transientShellWidgetClass,
    topLevel,
    NULL,
    0
    );

menubox = XtCreateManagedWidget(
    "menubox",          /* widget name   */
    boxWidgetClass,     /* widget class */
    pshell,             /* parent widget*/
    NULL,               /* argument list*/
    0                   /* arglist size */
    );

menulabel = XtCreateManagedWidget(
    "menulabel",        /* widget name   */
    labelWidgetClass,   /* widget class */
    menubox,            /* parent widget*/
    NULL,               /* argument list*/
```

Example 12-1. xmenu1: complete code (continued)

```
        0                   /* arglist size */
        );

    for (i = 0; i < 10; i++) {
        sprintf(buf, "menupane%d", i);
        menupane[i] = XtCreateManagedWidget(buf,    /* widget name    */
                commandWidgetClass, menubox, NULL, 0);

        XtAddCallback(menupane[i], XtNcallback, PaneChosen, i);
    }

    XtAppAddActions(app_context, trial_actions, XtNumber(trial_actions));

    XtAddCallback(quit, XtNcallback, Quit, 0);

    XtRealizeWidget(topLevel);

    XtAppMainLoop(app_context);
}
```

The `PlacePopup` action just places the pop up slightly to the left and above the position where the pointer button that popped it up was clicked, using the coordinates reported in the button event. The offset of ten pixels from the pointer position simply helps to make sure that the pointer is inside the menu.* Remember that the window created by a pop-up shell widget is a child of the root window and therefore is placed relative to the root window. The `ButtonPress` event pointer coordinates relative to the root window are used.

The `PaneChosen` callback function is a stub function used in this example as the notify callback for all the menu panes. In this example, it simply prints the name of the chosen pane to *stdout* and then pops down the menu using `XtPopdown`. In a real application, a different callback function would probably be registered for each pane.

Instead of calling `XtPopdown` in each of these separate callback functions, you could write a single additional callback function that calls `XtPopdown`, and then add it to the callback list for the Command widget that makes up each menu pane.

As usual, the pop up is created by first creating a pop-up shell, then a Box widget as its child, and then a series of Label and Command widgets as children of Box. The pop-up shell and the box are invisible. As with all menus, what you actually see is the array of children.

Note that this program does not include any code that would pop up the menu. We've done that from the application-defaults file shown in Example 12-2. The translations we have defined for the Label widget invoke Xt's built-in `XtMenuPopup` action.

Example 12-2. XMenu1: the application-defaults file

```
!
! Appearance Resources
!
*quit.label:    Quit
```

*It is important to provide a consistent user interface, so you should use the same offset in all menus. Menus in commercial widget sets such as the OPEN LOOK widgets have carefully designed and documented policies about pop-up window placement. This allows the user's "pointer reflexes" to be trained, so that using menus becomes as automatic and easy as possible.

Example 12-2. XMenu1: the application-defaults file (continued)

```
*label.label:    This is a Pretend Main Window; Press in here.
*menulabel.label:    MAIN MENU
!
! make all entries in menu same width
! (needs adjusting for longest entry)
!
*menulabel.width:    135
*menubox.Command.width:  135
!
! Pane Strings
!
*menupane0.label:  View Next
*menupane1.label:  View Previous
*menupane2.label:  Delete
*menupane3.label:  Move
*menupane4.label:  Copy
*menupane5.label:  Unmark
*menupane6.label:  View In New
*menupane7.label:  Reply
*menupane8.label:  Forward
*menupane9.label:  Use As Comp
!
! make Box leave no space around Command widgets in menu
!
*pshell.Box.hSpace: 0
*pshell.Box.vSpace: 0
!
! Functional Resources
!
*menubox.Command.translations:\
   <EnterWindow>:     highlight()              \n\
   <LeaveWindow>:     reset()                  \n\
   <BtnUp>:          set() notify() unset()
*label.translations:\
    <BtnDown>: placeMenu() XtMenuPopup(pshell)
*pshell.translations:\
    <BtnUp>: XtMenuPopdown(pshell)
```

There are a number of settings designed to give the Box widget a characteristic menu appearance: all of the Command widgets are forced to have the same size (rather than the size of their label), and the Box widget is forced to leave no space between the command widgets.

However, it is the new translations that are the critical part of this application-defaults file. The Label widget must be given translations so that a button press will pop up the widget. The supplied translation maps a button press into a call to the application action `Place-Popup` and then to Xt's predefined `XtMenuPopup` action. The argument to `XtMenu-Popup` in the translation is the instance name of the pop-up shell. Xt converts this string name into the widget ID of the pop-up shell before it can pop up the widget.

We are replacing rather than overriding this widget's translations because it is a Label widget and has no default translations.

The menu panes are Command widgets, but we need to adjust their event response so that they will be triggered on a button release with no corresponding press (since the press that

popped up the menu occurred in the application main window). We are replacing their translations to get rid of the translation for `ButtonPress` (which would still be present if we used the `#augment` directive, and we would have to create an action that did nothing in order to replace it with `#override`). The translation for `ButtonRelease` (abbreviated `BtnUp` in the translation table) calls all the actions that usually occur in Command widgets with both press and release.

Perhaps least obvious is the translation we have added to pop down the menu when the pointer button is released outside the menu. As mentioned earlier, Xt makes a passive global pointer grab on the pop-up shell (`pshell`) in the `XtMenuPopup` action. When the pointer is inside the menu, the Command widgets intercept these grabbed events, because they are descendants of `pshell` and they have a translation for `ButtonRelease` events. This invokes the actions in the selected Command widget. But when the pointer is outside the menu, the grabbed events are sent directly to the widget that was specified in the grab call, namely `pshell`. Therefore, the translation to pop down the menu on button release must be added to `pshell`. (Again, this translation table is simply replaced because the pop-up shell normally has no translations.)

12.2.2 A Drop-Down Menu

What are the desired characteristics of a drop-down menu? There is a Command widget or the like permanently visible in the application, with a label indicating some common characteristic of the items in the menu. When a button is pressed in this widget, the menu should pop up on or just below the button. Dragging the pointer down through the menu with the button still held should highlight the entry that is pointed to. Releasing the button in an entry should invoke or set that entry and pop down the menu. Moving out of the menu should not change this behavior, except that if the button is released anywhere outside of a menu pane, the menu should pop down without executing any entry.

If you compile and run *xmenu2* you can try out this style of menu. The appearance of this application is shown in Figure 12-4.

Invoking a menu as a drop-down is a simple enhancement of the spring-loaded invocation method just shown. We can do everything exactly the same as in the spring-loaded example, except that a drop-down menu should appear just below the `pressme` widget, not at the pointer position. Therefore, all we need to change is the placement code. However, since the coordinates in the event are not necessary for placing the pop ups, we can use a callback function instead of an action to place the pop up. (In general, it is better to use an existing callback than to add an action to do the same thing.)

Pop-up shell widgets have `XtNpop upCallback` and `XtNpopdownCallback` callback resources; the functions on these callback lists are called whenever the pop up is popped up or down using any of the Xt mechanisms.

In the last example we created an action called `PlaceMenu`, to move the pop-up shell before it was actually popped up. We included it in a translation along with the standard action `XtMenuPopup`, which was actually used to pop up the widget. *xmenu2* also uses the standard action `XtMenuPopup` to pop up the widget, but it uses the `XtNpopup-Callback` resource to provide the code to place the widget. Using the callback saves

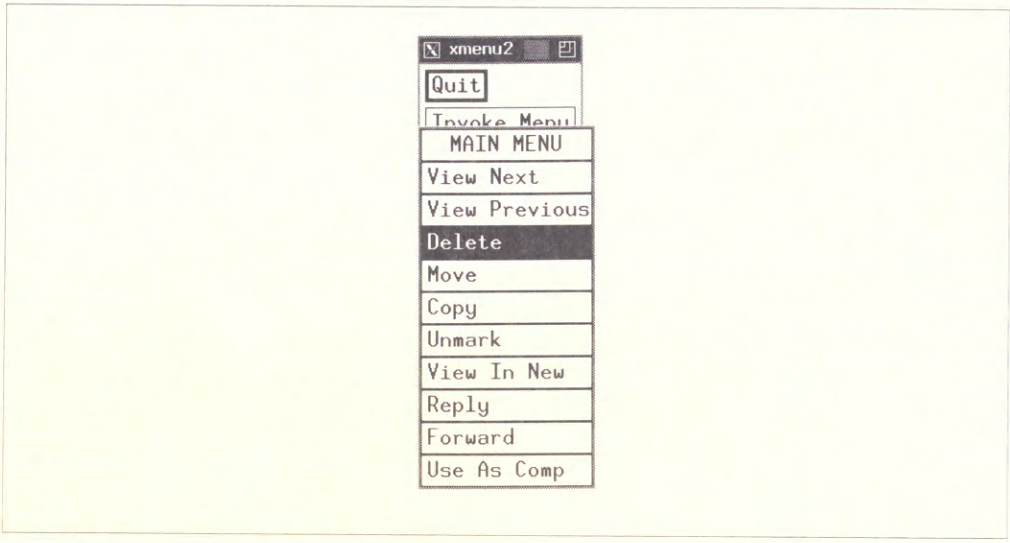


Figure 12-4. xmenu2: a drop-down menu

having to reference the placement action in the translation table. This is preferable, since the placement code should almost always be hardcoded rather than user-configurable. Another advantage of the pop-up and pop-down callbacks is that you may arrange for a pop up to be popped up or down in more than one way, and it may be convenient to have certain code called automatically in all cases.

(You can also use the `XtNpopupCallback` resource to specify a callback function to create a pop-up widget the first time it is popped up, instead of at application startup. The one problem is that the functions on the callback list are invoked every time the widget is popped up. To make sure that your function callback creates the pop up only once (the first time), the callback function should remove itself from the callback list by calling `XtRemoveCallback`.)

There is not enough difference between *xmenu1* and *xmenu2* to merit showing the complete code. All we have done is changed the `PlaceMenu` function from an action into a callback and changed its placement logic to place the pop-up relative to the invoking Command widget. We have then modified the application-defaults file accordingly.

Example 12-3 shows the `PlaceMenu` routine (now a callback, not an action) and the code to register it as a callback.

Example 12-3. xmenu2: code to place drop-down menu

```
/*ARGSUSED*/
void PlaceMenu(w, client_data, call_data)
Widget w;
XtPointer client_data;
XtPointer call_data;
{
    Position x, y;
```

Example 12-3. xmenu2: code to place drop-down menu (continued)

```
    Dimension height;

    /*
     * translate coordinates in application top-level window
     * into coordinates from root window origin.
     */
    XtTranslateCoords(pressme,    /* Widget */
            (Position) 0,         /* x */
            (Position) 0,         /* y */
            &x, &y);              /* coords on root window */

    /* get height of pressme so that menu is positioned below */
    XtVaGetValues(pressme,
            XtNheight, &height,
            NULL);

    /* move popup shell one pixel above and left of this position
     * (it's not visible yet) */
    XtVaSetValues(pshell,
            XtNx, x - 1,
            XtNy, y + height,
            NULL);
}

main(argc, argv)
int argc;
char **argv;
{
      .
      .
      .
    XtAddCallback(pshell, XtNpopupCallback, PlaceMenu, NULL);
      .
      .
      .
}
```

Xt calls the functions on the `XtNpopupCallback` list before it pops up the widget. This means that `PlaceMenu` is called before the `XtMenuPopup` action, placing the widget before it is popped up.

Note that the `XtTranslateCoords` routine determines the root window coordinates at the origin of the `pressme` widget. Because of the reparenting done by most window managers, this information cannot be obtained by using `XtGetValues` to read the `XtNx` and `XtNy` resources.*

*Incidentally, `XTranslateCoordinates`, the Xlib equivalent of `XtTranslateCoords`, gets the same information and a little more by querying the server. `XtTranslatCoords` does not have to make a server request because Xt stores this data locally. Each time a window in this application is moved, Xt receives this information as an event and updates its knowledge of the position of each window. This is an important optimization, because server queries are subject to network delays and tend to slow applications.

Example 12-4 shows the translation portion of the application-defaults file.

Example 12-4. XMenu2: translation portion of the application-defaults file

```
!
! Translation resources
!
*pressme.translations:\
   <EnterWindow>:      highlight()                \n\
   <LeaveWindow>:      reset()                    \n\
   <BtnDown>:          set() XtMenuPopup(pshell) reset()
!
*pshell.translations:\
   <BtnUp>:            XtMenuPopdown(pshell)
!
*menubox.Command.translations:\
   <EnterWindow>:      set()                      \n\
   <LeaveWindow>:      unset()                    \n\
   <BtnUp>:            notify() unset()
```

These translations are different from those for *xmenu1* only in that `pressme` is a Command widget, which already has its own translation table, rather than a widget without existing translations such as Label (which we used as a fake main window). We have modified the translations of `pressme` to be suitable for this use. Note that the translation no longer calls `PlaceMenu` as an action because it is now a callback.

The translations for the menu pane Command widgets are also somewhat different from *xmenu1*, but only for cosmetic reasons. This iteration of the *xmenu* example uses the `set` and `unset` actions instead of `highlight` and `reset` or `unhighlight` to make the Command widgets highlight their entire box instead of just an area near the border. (Although this modification makes the menu look more like a typical menu, it also seems to make it slower.)

To create several menus you simply need to replicate the code shown here, changing the variable names for each menu. The passive global grabs invoked by Xt for each menu do not interfere with each other even if they specify the same key/button combination, because they specify different windows in which the key/button combination will begin the grab.

It is sometimes useful to be able to get a list of children in a menu, especially if you add and subtract menu entries. You can do this by querying the *XtNchildren* and *XtNnumChildren* resources of the parent. The value of *XtNchildren* is a list of the widget ID's of the children. This technique allows you to eliminate maintaining global variables for every pane of every menu.

12.2.3 Cascaded Menus

A cascaded menu is a menu in which one or more panes do not invoke functions but instead bring up additional menus.

The techniques used to bring up cascaded menus can also be used to have dialog boxes bring up other dialog boxes. However, cascaded menus are more challenging because they rely on

the passive global pointer grab to receive the `ButtonRelease` event that occurs outside the menu and application.

You can implement a cascaded menu the same way for both spring-loaded and drop-down menus, simply by adding to the code we've already written to implement a single menu. We'll show you *xmenu5*, the spring-loaded version, since it is slightly shorter. (*xmenu4* is the equivalent drop-down version.) Both are included in the example source code. In this example, only one menu pane will be used to invoke a submenu. However, this technique can be generalized to have additional panes bring up additional submenus.

First, let's describe exactly how we expect the cascaded menu to work. Figure 12-5 shows both menus popped up. (Compile the program and try it.)

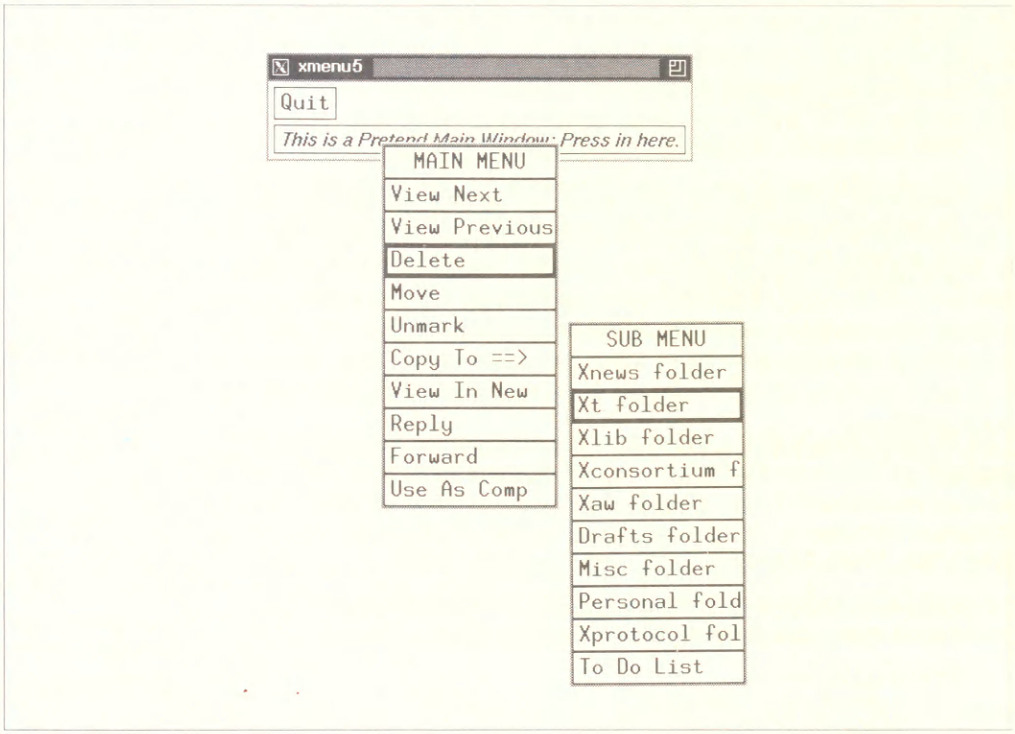


Figure 12-5. xmenu5: cascaded spring-loaded menus

The main menu works as described above. However, one of the panes—the one that brings up the submenu—has an arrow pointing to the right after its label. This pane does not highlight itself when the pointer moves inside (telling the user that this pane is different). Instead, when the user moves the pointer out through the right edge of the pane, the submenu pops up. The submenu operates just like the main menu. When the button is released inside either menu, the callback function associated with the chosen pane will be invoked. When the button is released outside of either menu, both menus pop down. If the pointer is moved back out of the submenu into the main menu, only the submenu pops down.

To create the submenu, we create a new pop-up shell, Box widget, and a set of Command widgets, and add callbacks for each function the submenu panes will invoke (in this example, one common callback). Then we write three actions: `PlaceMenu` (which you have already seen), `CheckRightAndPopupSubmenu` (which places and pops up the submenu if the pointer leaves the main menu pane through its right side), and `PopdownSubmenu` (which pops down the submenu if the pointer leaves the submenu). These actions are shown in Example 12-5.

Example 12-5. xmenu5: actions that place, pop up, and pop down main menus and submenus

```
/*ARGSUSED*/
void PlaceMenu(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    XButtonEvent *bevent = (XButtonEvent *) event;

    /* should make sure coordinates allow menu to fit on screen */

    /* move submenu shell to slightly left and above button
     * press position */
    XtVaSetValues(pshell,
            XtNx, bevent->x_root - 10,
            XtNy, bevent->y_root - 10,
            NULL);
}

/*ARGSUSED*/
void CheckRightAndPopupSubmenu(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    XLeaveWindowEvent *leave_event = (XLeaveWindowEvent *) event;
    Dimension height, width;

    XtVaGetValues(w,
        XtNheight, &height,
        XtNwidth, &width,
        NULL);

    if ((leave_event->x > width) && (leave_event->y > 0)
            && (leave_event->y < height)) {
        /* move submenu shell to start just right of pane,
         * using an arbitrary offset to place pointer in
         * first item. */
        XtVaSetValues(subshell,
                XtNx, leave_event->x_root,
                XtNy, leave_event->y_root - 12,
                NULL);
        XtPopup(subshell, XtGrabNonexclusive);
    }
}

/*ARGSUSED*/
```

Example 12-5. xmenu5: actions that place, pop up, and pop down main menus and submenus (continued)

```
void PopdownSubmenu(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    XtPopdown(subshell);
}
```

As usual, the application-defaults file specifies which events trigger these actions. We'll show this file in a moment, but for now you need to know just that `CheckRightAndPopupSubmenu` and `PopdownSubmenu` are triggered by `LeaveNotify` events. We want the submenu to pop up and down only when the pointer leaves through certain parts of certain sides of the widget—the entire right side of a pane for popping it up, and the part of the submenu touching the main menu on the left side for popping it down. These two actions are called in response to all `LeaveNotify` events, and they check if the pointer left through the correct parts before popping up and down the submenu.

As you may recall, we mentioned earlier that no matter what the arguments, `XtPopup` and all other Xt facilities for grabbing, except `XtMenuPopup` and `XtPopupSpringLoaded`, make no passive global grab, and therefore can't be used for spring-loaded or drop-down main menus. It turns out that for submenus the opposite is true—`XtPopup` and `XtCallback*` work fine, but `XtMenuPopup` is inappropriate because no new passive grab is needed. The original grab directs events normally to all widgets in the application, including the submenu, and directs all events that occur outside the application to `pshell`.

The `CheckRightAndPopupSubmenu` action calls `XtPopup` with a grab mode of `XtGrabNonexclusive`. This grab mode controls Xt's event dispatching within the application—it has nothing to do with the passive global grab that Xt makes from the `XtMenuPopup` action. `XtGrabNonexclusive` means that widgets in the cascade but outside of the submenu will continue to get events normally. The grab mode specified in the call to `XtPopup`, or specified by the standard pop-up callback function selected (`XtCallbackExclusive` or `XtCallbackNonexclusive`) merely control the event dispatching within the application.

As an exercise, you may want to modify the example so that `CheckRightAndPopupSubmenu` calls `XtPopup` with a grab mode of `XtGrabExclusive`, and see how the menus work as a result. The `XtGrabExclusive` mode means that only the most recent pop up popped up will get events, while `XtGrabNonexclusive` means that all pop ups in a popped-up cascade will get events. In this case, when the submenu is popped up, it alone will get pointer events if you use grab mode `XtGrabExclusive`, while both it and the main menu will get pointer events if you use grab mode `XtGrabNonexclusive`. Because of the logic that pops down the submenu when the pointer leaves it through the portion adjoining the main menu, you can see this difference only if you move out through another part of the submenu and then around into the main menu again. (In the example code distribution, *xmenu5.c* uses `XtGrabNonexclusive` and *xmenu4.c* uses `XtGrabExclusive`, so that you can compare the results of these two flags.)

The user-interface conventions for a particular widget set usually specify which kinds of pop ups should have exclusive grabs and which nonexclusive. Note that the effect of these two grab modes is the same unless there is more than one pop-up widget in a cascade visible.

`CheckRightAndPopupSubmenu` places the submenu itself (instead of using a separate action) because the menu should be placed only when it is first popped up. If the placement code were a separate action, it would be called every time a `LeaveWindow` event arrived, even if not through the correct border of the widget. (Remember that this code is in an action rather than a callback because it uses the contents of the event.)

The translation portion of the application-defaults file for *xmenu5* is shown in Example 12-6.

Example 12-6. XMenu5: translation portion of application-defaults file

```
!
! Appearance resources
!
*menupane5.label:  Copy To ==>
   .
   .    (other appearance resources not shown)
   .
!
! Translation resources
!
!  popping down both menus
*pshell.translations:\
    <BtnUp>: XtMenuPopdown(subshell) XtMenuPopdown(pshell)
!
!  popping up main menu
*label.translations:\
    <BtnDown>: placeMenu() XtMenuPopup(pshell)
!
!  popping down submenu
*menubox.menupane5.translations:\
   <LeaveWindow>:     checkRightAndPopupSubmenu()
!
! Main Menu translations
*menubox.Command.translations:\
   <EnterWindow>:     highlight()              \n\
   <LeaveWindow>:     reset()                  \n\
   <BtnUp>:          set() notify() unset()
!
! Sub Menu translations
*subbox.translations:\
   <LeaveWindow>:     popdownSubmenu(subbox)
*subbox.Command.translations:\
   <EnterWindow>:     highlight()              \n\
   <LeaveWindow>:     reset()                  \n\
   <BtnUp>:          set() notify() unset()
```

The first three translation tables handle popping up the main menu and making the menu Command widgets work as expected. We've seen these in previous examples.

The translation table for `pshell` pops down one or both menus; no error or warning is caused if only the main menu is up. This translation table works because the button release is

sent to `pshell` if it occurs outside a menu regardless of whether just the main menu or both menus are up.

In this case, menu pane 3 is the pane that will pop up the submenu. The label for this pane is shown at the top of Example 12-6. The translation for this pane replaces all the normal translations for highlighting and notifying with a single translation for `LeaveWindow` events. These events in this widget trigger the `CheckRightAndPopupSubmenu` action which has already been described.

The translations for the `subbox` widget invoke `PopdownSubmenu` action to check whether the pointer left the submenu. In this case, the submenu is popped down, but the original pop up remains visible.

The pane that will pop up the submenu and its event handling characteristics is controlled from the application-defaults file. Therefore, it may seem like you can change which menu pane invokes the submenu simply by changing the application-defaults file. This is true here, because the menu actions are nonfunctional and simply call a common callback. But it would not be possible if each pane invoked its own callback. To give the user the freedom to rearrange the menu, you would have to use actions instead of callbacks.

Note that you can define accelerators for any of the menus shown up to this point simply by placing settings for the XtNaccelerators resource in the application-defaults file. You would set this resource for every Command widget in the menus. This provides keyboard shortcuts for popping up the menu and choosing a pane. However, remember to make sure that each key combination is unique.

12.2.4 Using the R4 SimpleMenu Widget

Once you have R4, you can use a real menu widget: the R4 SimpleMenu widget. The internals of this widget and its children, which are gadgets, will be described later. This section describes a simple application that uses the SimpleMenu widget.*

R4 supplies three types of panes to be used with the SimpleMenu widget: SmeBSB (an entry composed of a bitmap, a string, and another bitmap), SmeLine (a horizontal line between entries), and Sme (a blank entry). The SimpleMenu widget is itself a subclass of the pop-up shell widget, and therefore no separate pop-up shell needs to be created.

R4 also provides a MenuButton widget, which is a subclass of Command with built-in placement and pop-up code. Using a MenuButton to invoke a menu makes it even simpler to implement a drop-down menu, since the pop up and placement code can be eliminated from the application.

The *xmenu7* application shown in Example 12-7 creates a SimpleMenu widget with panes of all three types. The menu is invoked in drop-down style using the R4 MenuButton widget.†

*Note that if you have an R3 version of the Xt Intrinsics library, you cannot use the SimpleMenu widget even if you can get its code. The widget depends on changes to Xt introduced in R4.

†This example was written by Chris Peterson of MIT Project Athena and modified only slightly by the authors.

As an additional enhancement, the menu marks or unmarks each item when it is selected in addition to calling a callback function. This iteration marks entries with the X logo, which is available as a standard bitmap in */usr/include/X11/bitmaps* (on UNIX systems).

Figure 12-6 shows the appearance of the program.

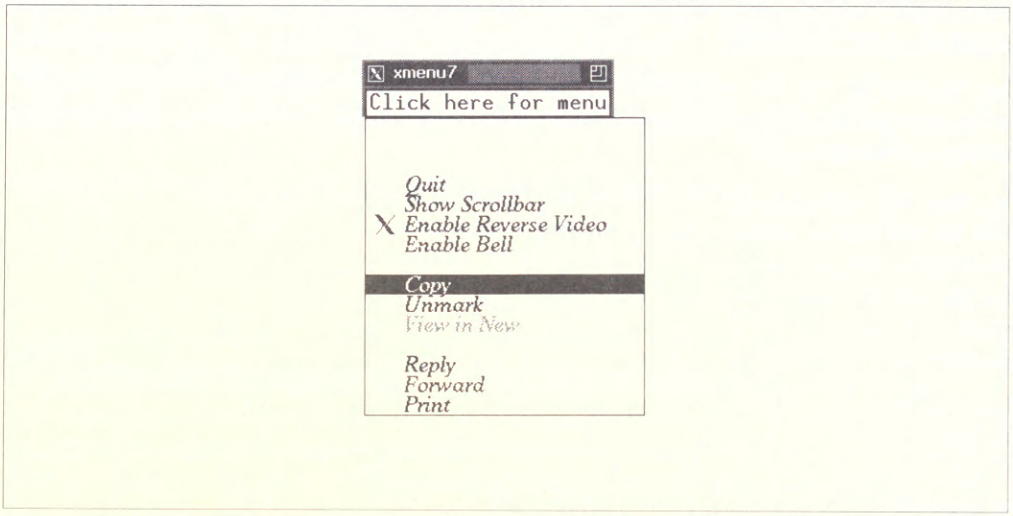


Figure 12-6. xmenu7: a menu using the Athena SimpleMenu widget

Example 12-7. xmenu7: using the SimpleMenu widget and its children

```
/*
 * xmenu7.c
 */

#include <stdio.h>

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/bitmaps/xlogo16>

#include <X11/Xaw/MenuButton.h>
#include <X11/Xaw/SimpleMenu.h>
#include <X11/Xaw/SmeBSB.h>
#include <X11/Xaw/SmeLine.h>

#define NUM_MENU_ITEMS 12

static String menu_entry_names[] = {
  "quit",
  "item1",
  "item2",
  "item3",
  "line",
  "item5",
  "item6",
  "item7",
  "blank",
  "menu1",
  "menu2",
```

Example 12-7. xmenu7: using the SimpleMenu widget and its children (continued)

```
  "menu3",
};

static Boolean status[NUM_MENU_ITEMS];
static Pixmap mark;

/* ARGSUSED */
static void
MenuSelect(w, client_data, garbage)
Widget w;
XtPointer client_data;
XtPointer garbage;          /* call_data */
{
    int pane_num = (int) client_data;

    printf("Menu item %s has been selected.\n", XtName(w));

    if (pane_num == 0)      /* quit selected. */
        exit(0);

    if (status[pane_num])
        XtVaSetValues(w,
                XtNleftBitmap, None,
                NULL);
    else
        XtVaSetValues(w,
                XtNleftBitmap, mark,
                NULL);

    status[pane_num] = !status[pane_num];
}

void
main(argc, argv)
char **argv;
int argc;
{
    XtAppContext app_context;
    Widget topLevel, menu, button, entry;
    int i;
    Arg arglist[1];

    topLevel = XtVaAppInitialize(
        &app_context,       /* Application context */
        "XMenu7",           /* Application class */
        NULL, 0,            /* command line option list */
        &argc, argv,        /* command line args */
        NULL,               /* for missing app-defaults file */
        NULL);              /* terminate varargs list */

    button = XtCreateManagedWidget("menuButton",
            menuButtonWidgetClass, topLevel,
            arglist, (Cardinal) 0);

    menu = XtCreatePopupShell("menu", simpleMenuWidgetClass,
            button, NULL, 0);

    for (i = 0; i < NUM_MENU_ITEMS ; i++) {
        String item = menu_entry_names[i];
```

Example 12-7. xmenu7: using the SimpleMenu widget and its children (continued)

```
        if (i == 4)    /* use a line pane */
            entry = XtCreateManagedWidget(item,
                    smeLineObjectClass, menu,
                    NULL, 0);
        else if (i == 8) /* blank entry */
            entry = XtCreateManagedWidget(item, smeObjectClass, menu,
                    NULL, 0);
        else {
            entry = XtCreateManagedWidget(item, smeBSBObjectClass,
                    menu, NULL, 0);

            XtAddCallback(entry, XtNcallback, MenuSelect, (XtPointer) i);
        }
    }

    mark = XCreateBitmapFromData(XtDisplay(topLevel),
            RootWindowOfScreen(XtScreen(topLevel)),
            xlogo16_bits, xlogo16_width, xlogo16_height);

    XtRealizeWidget(topLevel);
    XtAppMainLoop(app_context);
}
```

You will notice that each pane has an `XtNleftBitmap` resource, which is alternately set to the X logo or to nothing each time that item is selected.

The application-defaults file for *xmenu7* is shown in Example 12-8.

Example 12-8. XMenu7: application-defaults file

```
!
! For Color workstations only.
!
XMenu7*SimpleMenu*foreground:              SteelBlue
XMenu7*SimpleMenu*menuLabel.foreground:     Gold
XMenu7*SimpleMenu*line.foreground:         Grey

XMenu7*MenuButton.label:         Click here for menu

XMenu7*SimpleMenu*menuLabel.vertSpace:         100
XMenu7*SimpleMenu*menuLabel.leftMargin:         70
XMenu7*SimpleMenu.label: Main Menu
XMenu7*SimpleMenu*item1*label: Show Scrollbar
XMenu7*SimpleMenu*item2*label: Enable Reverse Video
XMenu7*SimpleMenu*item3*label: Enable Bell
XMenu7*SimpleMenu*item4*label: Disable Auto-Repeat
XMenu7*SimpleMenu*item5*label: Copy
XMenu7*SimpleMenu*item6*label: Unmark
XMenu7*SimpleMenu*item7*label: View in New
XMenu7*SimpleMenu*menu1*label: Reply
XMenu7*SimpleMenu*menu2*label: Forward
XMenu7*SimpleMenu*menu3*label: Print
XMenu7*SimpleMenu*quit*label:           Quit
XMenu7*SimpleMenu*RowHeight:         16
XMenu7*SimpleMenu*item7*sensitive:      off
XMenu7*SimpleMenu*HorizontalMargins:     30
! Just for fun:
```

Example 12-8. XMenu7: application-defaults file (continued)

```
XMenu7*font: *times*medium*18*iso8859-1
!XMenu7*item7*font: *helv*medium*24*iso8859-1
!XMenu7*item8*font: *helv*bold*24*iso8859-1

*quit*accelerators:\
    <Key>q: notify()
*item1*accelerators:\
    <Key>1: notify()
*item2*accelerators:\
    <Key>2: notify()
*item3*accelerators:\
    <Key>3: notify()
*item5*accelerators:\
    <Key>5: notify()
*item6*accelerators:\
    <Key>6: notify()
*item7*accelerators:\
    <Key>7: notify()
*menu1*accelerators:\
    <Key>m: notify()
```

This file simply sets various cosmetic features of the menu. (See Appendix C, *Naming Conventions*, for information on font-naming conventions.) Naturally, you could easily set the strings for each menu entry in this file. Note that there are no translation tables in this file because MenuButton and SimpleMenu are doing exactly what they were designed to do.

Accelerators can be defined for menus with gadget children, but not in the usual sense. They cannot be defined to invoke the actions of the gadget children, but they can invoke global application actions, which for menus is usually good enough. For example, in the R4 *xmh*, one item on one of the menus incorporates new mail. From the widget, the `notify` action of the menu pane gadget calls the `DoIncorporateNewMail` callback function. The `XtNaccelerators` resource for the SimpleMenu widget itself (not the gadgets) maps a Meta-I key event into a call to the `XmhIncorporateNewMail` global action. `XmhIncorporateNewMail` then calls `DoIncorporateNewMail`. This use of accelerators depends on having both a callback and an action form of each function.

12.2.5 Delayed Popup Creation

As we've seen, a pop up may consist of a single widget, or it may be a shell widget which contains a composite widget which contains a number of children. In the latter case, creating all those widgets (or gadgets) takes time. It may be beneficial to create those widgets using idle time in the application instead of delaying startup. In either case, it may make sense to create the pop up only when it is needed, to minimize wasted resources.

If your goal is to speed startup, and you want all menus created even if some are never used, you can register a work procedure to create each pop up, as described in Section 8.5. As you may recall, a work procedure uses idle time in the application to call a function, which must return swiftly. If you use this technique, you add one work procedure for each pop up you need to create, and you need to add code to make sure that the pop up has been created before the user is allowed to use it.

If your goal is to create only the required pop ups, you can create the pop up in a callback function or action routine that you have registered to place or pop up the pop up. In this case, you would have a static variable in the callback or action to make sure that the pop-up widgets are only created the first time the pop up is popped up. You need to have created the pop-up shell before this can work.

There is also another way to create only the required pop ups. Shell widgets have an `XtNcreatePopupChildProc` resource which you can set to a function that creates the Shell's children. See `XtCreatePopupChildProc(2)` in Volume Five, *X Toolkit Intrinsics Reference Manual*, for the calling sequence of this function type. The Xt specification does not say whether an `XtCreatePopupChildProc` is called just once when the shell is first popped up or every time it is popped up. But the MIT implementation of Xt calls it every time the shell is popped up, so you will again need a static variable to make sure the children are only created once.

12.3 About Dialog Boxes

Although we have been talking so far exclusively about menus, much that has been said is also true of dialog boxes. Both menus and dialog boxes that get user input usually need to get that input before other application functions can be invoked. Of course, one way to disable all other application functions is to make all other widgets insensitive with `XtSetSensitive` (passing it `FALSE`). Setting the sensitivity of one common ancestor does this efficiently, but even this is too slow because all the widgets redraw themselves dimmed or grayed. It is much faster to use a global grab. Unlike menus, which require the global grab in order to get button release events outside the application so they can pop down properly, dialog boxes do not, strictly speaking, need a grab. But they sometimes make the grab anyway to disable other application functions.

Dialog boxes can also invoke other dialog boxes. For example, a dialog box that gets input might check the validity of the input before popping down the dialog, and if incorrect, pop up a message telling the user the problem with the input. Cascaded dialog boxes are implemented the same way as cascaded menus. Note that, as a general rule, sub-dialog boxes are popped up with grab mode `XtGrabExclusive`, which means that the user must satisfy the most deeply nested dialog first.

Some pop ups do not need to disable other application functions. For example, imagine a dialog box that informed the user of some fact without requiring confirmation. This kind of pop up would be popped up with grab mode `XtGrabNone`, allowing the user to continue with other application functions.

We pointed out earlier that the built-in callback functions for popping up a widget are not useful for menus because they make no passive global pointer grab. However, they come in handy for dialog boxes. The functions `XtCallbackNone`, `XtCallbackExclusive`, and `XtCallbackNonexclusive` can be used to pop up dialog boxes, as long as the position of the dialog box need not depend on information in an event.

We haven't shown how to use Xt's standard callback for popping down a widget: `XtCallbackPopdown`. Instead of calling `XtPopdown` in the callback functions for each menu entry, we can add `XtCallbackPopdown` to the callback list after the existing callback function. `XtCallbackPopdown` requires an `XtPopdownId` structure to be passed as the *`client_data`* argument. This structure must contain the pop-up shell and the widget that invoked the pop up (the MenuButton or Command widget).

All three of the standard pop-up callbacks set the invoking widget to insensitive mode before popping up the widget. `XtCallbackPopdown` resets the invoking widget to sensitive mode. Therefore, if you use `XtCallbackNone`, `XtCallbackNonexclusive`, or `XtCallbackExclusive` without also using `XtCallbackPopdown`, remember to set the widget to sensitive mode yourself. This feature is useless but also harmless when the pop up is spring-loaded, because the invoking widget is often the main application window and that widget rarely responds to sensitivity.

In certain rare cases, you may want to use `XtAddGrab` and `XtRemoveGrab` directly to append a widget to or remove a widget from the current pop-up cascade. These functions are called internally by the Xt facilities that pop widgets up and down, and should not be necessary on their own. Note that these functions never make a request to the server to start or release a passive global pointer grab—they affect only Xt's internal event dispatching. (However, the functions `XtGrabKey`, `XtGrabKeyboard`, `XtGrabButton`, and `XtGrabPointer` do initiate global grabs. These functions are described in Chapter 13, *Miscellaneous Toolkit Programming Techniques*.)

12.4 Gadgets

When an application includes many different menus with many fields each, the overhead of having separate widgets for every menu pane becomes significant. Because each widget requires structures on the client side and windows on the server side, every widget increases the executable size and server memory usage and increases the traffic over the network. It is always a good idea to minimize the number of widgets used in your application.

Writing a single widget that implements an entire menu, including all its panes, solves this problem. The widget could define subresources for configuring each pane. (The callback list would be one of these subresources. `XtCallCallbacks` could not be used to invoke these callbacks because it cannot distinguish between subparts. However, the callbacks can be called directly by looping through the callback list. This is awkward, but not too difficult.) This widget would not lack flexibility if the subparts are implemented as completely separate code so that the types of menu panes are extensible. However, gadgets turn out to be an easier and more elegant solution to the problem.

As of Release 4, Xt provides gadgets, which are simplified widgets that do not create windows.* Gadgets require less memory than widgets on the client side (and less disk space for the executable file) and consume none at all on the server side. Gadgets can be used for the

*The OSF Motif 1.0 Intrinsics support a different implementation of gadgets than those described here. Motif 1.1, however, uses the R4 standard Intrinsics, which includes the gadget implementation described here.

panes of a menu, solving all of the problems just discussed. A gadget is fully configurable using the resource database just like a widget, and can have its own callback list.

However, the reduced consumption of gadgets does have a price. Gadgets have to draw on their parent's window, and they share this space with the parent and with all other gadget children. The gadgets and the parent must agree to draw in certain areas only. For this reason, gadgets must be used with a special composite widget parent that is prepared to manage them properly.

The Release 4 SimpleMenu widget is such a parent. It is a composite widget designed to manage gadget children. The gadgets provided by Release 4 are Sme, SmeBSB, and SmeLine (where Sme stands for Simple Menu Entry). These provide a blank entry (and generic menu entry superclass), an entry which can contain a bitmap, a string, and another bitmap (thus BSB), and an entry that draws a horizontal line. We will use and describe this widget and these gadgets both to show how menu widgets are built and to demonstrate how the parent and the gadgets work together.

Gadgets do not handle events automatically as widgets do, and because they have no windows, the server does not handle overlapping between them. This places certain demands on the parent. All the gadgets that are children of a particular parent share that parent's window. The parent is responsible for coordinating the gadget children, telling them about events by calling their functions. Therefore, the composite widget that manages a group of gadgets must be specially designed for that purpose, not a general-purpose composite or constraint widget such as Box or Form. It is possible for a composite widget to manage both gadget and widget children, but its code has to be more involved to do this.

Like normal widgets, gadgets provide their own code to redraw themselves in their `expose` method. However, since gadgets do not receive events, they depend on the parent to directly call their `expose` method. The parent keeps track of the geometry of each child, and when the parent's `expose` method is called, this method calculates whether the area exposed overlaps any of the gadget children. If the area exposed does overlap a gadget, the parent's `expose` method calls that gadget's `expose` method, which redraws the area.

A gadget's actions also have to work differently from widget actions because of the fact that gadgets don't get events. A gadget defines its actions as methods—as fields in its class part structure—instead of in an action list and translation table. It initializes these fields directly to pointers to functions during class initialization. The parent widget has corresponding actions that are defined and operate like normal actions, except that they determine which gadget the event that invoked the action occurred in and call the gadget method corresponding to that action. In other words, the parent has actions that operate the gadget children.

One weakness of a menu composed of gadget panes is that gadgets cannot have an accelerator table. Therefore, accelerators cannot be used to provide a keyboard equivalent that would invoke each menu pane.

The parent of gadgets has to position the gadget children so that they do not overlap, or take care of the consequences if they do overlap. Since the gadgets draw on the parent's window, if they did overlap they would draw over each other's graphics, with unpredictable results. The parent would have to calculate the area of overlap between two gadgets, and clear this area before letting one of the gadgets draw itself. (A gadget could clear its own area before drawing, but this would be unnecessary in many cases, and would cause flashing.)

Gadgets are subclasses of RectObj, one of the invisible superclasses of Core that we have so far ignored because for widgets it is safe to assume that Core is the top of the widget class hierarchy.* The actual class hierarchy leading up to Core is shown in Figure 12-7.

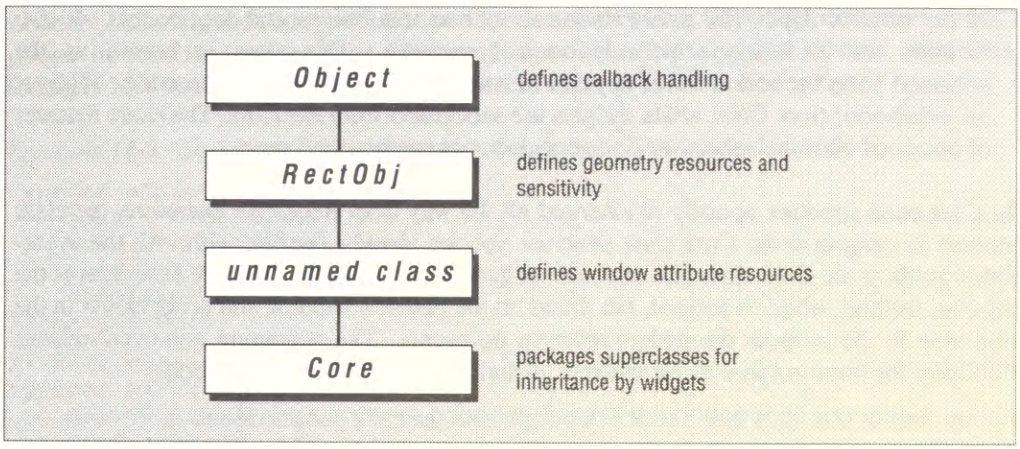


Figure 12-7. Class hierarchy derivation of Core

The "unnamed class" actually has a name (WindowObj) but this class is intentionally undocumented in the Xt specification so that its characteristics can be changed in later releases without compatibility problems. You should never create subclasses directly from the unnamed class.

The superclasses of Core are not real classes in the sense that they do not play by all the rules we have described in Chapter 5, *Inside a Widget*. For one thing, each shares what we have been calling the Core class structure instead of adding its own part structure. Applications are never intended to create instances of these superclasses—they are really just part of the implementation of Xt. Instead of developing all the characteristics of widgets in one large base class Core, it made more sense to implement Xt in object-oriented fashion by dividing the implementation into separate pseudo-classes. It is important to know what each early class defines simply so that you know what characteristics are available in gadgets and which are available only in widgets. Each class defines the following features:

- Object defines only the `XtNdestroyCallback` resource and the underlying support for callbacks in general.
- RectObj defines the geometry resources (`XtNx`, `XtNy`, `XtNwidth`, `XtNheight`, and `XtNborder_width`) and the resources that control sensitivity: `XtNancestorSensitive` and `XtNsensitive`. RectObj itself doesn't use the sensitivity resources. They are provided at this level in the hierarchy so that sensitivity can be set for gadgets. Gadgets draw themselves according to these resources (gray if insensitive),

*In R3, the header files for these invisible classes are not so invisible. They were all present in */usr/include/X11* with all the other Xt header files. In R4, only the header files for RectObj are public, so that you can write and compile gadgets.

and check these resources in their action routines (stored in their class part methods), invoking their callback function only if sensitive.

- The unnamed class (WindowObj) adds many window-oriented resources: ones that control window attributes such as the background pixmap, permanent window features such as the window depth, and event resources such as translations and accelerators. It also includes the `XtNmappedWhenManaged` resource. This class is known as the unnamed class because it never appears in widget, gadget, or application code. Widgets are subclassed from Core, while gadgets are subclassed from RectObj. The exact features of unnamed class are subject to change and should not be relied upon.

The Core class structure actually is inherited all the way from RectObj. Therefore, the class structure in gadgets is the Core class structure you are already familiar with. All the event-related fields in the Core class part structure of gadgets are unused. The only exception is the `expose` method, which is present, but draws on the parent's window and is not called in the usual way by Xt because the gadget receives no events. The remaining non-event-related fields have the same purpose as for widgets, including all the remaining methods.

Without further ado, let's take a look at a gadget, and then at a gadget parent.

12.4.1 Inside a Gadget

Many portions of a gadget's code are exactly the same as those of a widget, as described in Chapter 5, *Inside a Widget*. This section briefly summarizes the parts that are identical so that you know that nothing is left out, and describes the differences in detail.

The code for gadgets and widgets includes the same three implementation files with the same naming conventions. As in Chapter 5, we'll take the three implementation files one at a time, beginning with the private header file, then the code file, and then the public header file.

As in widget code, many of the conventions described here are automatically taken care of for you when you are writing a new gadget if you copy all three files of an existing gadget and then globally change the gadget class name.

The Athena menu pane gadgets are implemented in two class levels:

- Sme (Simple Menu Entry) defines the callback for an entry, the actions to highlight, unhighlight, and notify, and the code that allows subclasses to inherit or replace these actions (because they are defined as methods). The actual functions for highlight and unhighlight are empty, while the notify action calls the callback. The `expose` method of this gadget is also empty. This gadget can be used by itself to create a blank entry.
- SmeBSB and SmeLine are each subclasses of Sme. SmeLine replaces only the `expose` method of its superclass. SmeBSB replaces both the `expose` method and the highlight and unhighlight actions of the superclass. (Sme can be subclassed to create new types of menu entries.)

The following sections describe both Sme and SmeBSB.

12.4.2 Private Header File

The private header file for a gadget is identical in format to the private header file for a widget. It defines a class part structure for this class of gadget and then a complete class structure including the class parts of superclasses and this class. The only difference is that a gadget inherits its features from Object and RectObj while a widget inherits from Core. Example 12-9 shows the complete class structure of the R4 Athena Sme gadget.

Example 12-9. Sme gadget: class part and complete class structure declaration

```
typedef struct _SmeClassPart {
    void (*highlight)();
    void (*unhighlight)();
    void (*notify)();
    XtPointer extension;
} SmeClassPart;

/* Full class record declaration */
typedef struct _SmeClassRec {
    RectObjClassPart    rect_class;
    SmeClassPart  sme_class;
} SmeClassRec;

#define XtInheritHighlight   ((_XawEntryVoidFunc) _XtInherit)
#define XtInheritUnhighlight XtInheritHighlight
#define XtInheritNotify      XtInheritHighlight
```

Notice that the complete class structure declaration does not include the class part for the Object class, even though it is a superclass. This is because all the superclasses of Core share the same class part structure.

The class part structure for Sme defines three methods—these are essentially the gadget's actions, but they will be invoked by the gadget parent's actions, not directly by Xt. The `extension` field allows fields to be added to this structure in a future version while retaining binary compatibility. In a future version this field could be changed to point to an extension structure.*

Any class that defines methods must provide code to allow them to be inherited or replaced by subclasses. The Sme class therefore must define the `XtInherit` constants that allow the methods to be inherited. The *.c* code file provides the `class_part_init` method that allows them to be replaced. (See Section 11.4.5.)

In the private header file for SmeBSB, the class part structure would contain only the extension field, because SmeBSB will be using the highlight, unhighlight, and notify fields defined by Sme. However, the *.c* file will initialize these fields to point to its own functions.

*Note that the `extension` field is defined as type `XtPointer`. In R4, all occurrences of `caddr_t` have been replaced with `XtPointer`. On most systems, `XtPointer` will be defined to be `caddr_t`. But for some architectures, `caddr_t` is too small to hold a pointer to a function. On such systems, `XtPointer` will be defined to be larger. The `caddr_t` type will continue to work on most systems, but you are advised to use `XtPointer` instead for maximum portability.

The instance part structure and complete instance structure of Sme are shown in Example 12-10.

Example 12-10. Sme gadget: instance part and complete instance structure declaration

```
typedef struct {
    /* resources */
    XtCallbackList callbacks;    /* The callback list */
} SmePart;

typedef struct _SmeRec {
    ObjectPart     object;
    RectObjPart    rectangle;
    SmePart  sme;
} SmeRec;
```

The `SmePart` adds a `callbacks` resource. The complete `SmeRec` includes the prior elements in the widget hierarchy: `ObjectPart` and `RectObjPart`. Note that unlike the class structure, the Object class does appear in the complete instance structure, because the superclasses of Core do not share instance structures.

The instance part structure for SmeBSB includes the usual fields to maintain the graphics state of the entry, including the label, colors, font, GCs, and positioning information. The complete instance structure for SmeBSB is the same as the one for Sme but with the SmeBSBPart structure added at the end.

12.4.3 The Gadget Source File

The source file for a gadget is identical in form to a widget source file. The only differences are that the superclass of a gadget in the class structure initialization is `rectObjClassRec`, and the complete instance structure type is called `SmeObject` for a gadget where it would have been `SmeWidget` if the entry was a true widget. Therefore, `SmeObject` is the type into which you cast the pointer to the instance structure before accessing the structure's fields in all the widget methods.

In addition, several of the Core class structure fields that might be used in a widget are never used in gadgets. The following is the complete list of fields that are always initialized to a certain value in a gadget:

- `realize` set to `NULL`
- `actions` set to `NULL`
- `num_actions` set to 0 (zero)
- `compress_motion` set to `FALSE`
- `compress_exposure` set to `FALSE`
- `compress_enterleave` set to `FALSE`
- `visible_interest` set to `FALSE`

- `resize` set to `NULL`
- `display_accelerators` set to `NULL`

Setting these fields otherwise (of the right type) probably won't cause the gadget to crash, but won't accomplish anything useful either.

Gadgets, like widgets, should always define the `query_geometry` method, and either define `set_values_almost` or initialize it to `XtInheritSetValuesAlmost`. The remainder of the fields and methods have the same purpose and are used in the same way as for widgets.

There are, however, slight differences in the code for certain gadget methods. The `expose` method checks not only its own sensitivity but also its parent's sensitivity before deciding whether to draw the entry in normal colors or grayed. When creating GCs using `XCreateGC` or creating any other server resources from the `initialize` method using an Xlib call, you must remember to use the parent's window, since the gadget has no window. (`XtWindow` is not smart enough to give you the parent's window ID in the case of gadgets.) Also, the parent's resource values, such as `background_pixel`, may be used to provide data in common among all instances of a subclass like SmeBSB.

12.4.4 The Public Header File

The only difference in the public header file between widgets and gadgets is that what would have been `Widget` for a widget is `Object` for a gadget. As mentioned previously, if you are writing a gadget you should start by copying the files for an existing gadget and then globally change names. Then you will start with the proper conventions already in place.

12.4.5 The Gadget Parent

A gadget parent is a composite widget designed to manage gadget children. Gadget parents perform all the geometry management tasks that all composite widgets perform, described in Chapter 11, *Geometry Management*. Gadgets also follow all the rules of normal widget children. However, gadget parents also have the added responsibility of managing the overlap of gadgets or making sure they don't overlap, and of handling events for the gadgets and calling gadget code. This section describes the gadget-managing role of the gadget parent.

The Athena SimpleMenu widget is designed to manage the gadget children already described, Sme, SmeLine, and SmeBSB (and any other subclass of Sme that is written later). It forms a vertical menu with horizontal panes. It is quite a large widget because it contains all the geometry management code in addition to code for managing events for the gadgets. We'll concentrate just on the code that manages events for the gadgets, since the geometry management code is as described in Chapter 11, *Geometry Management*.

Let's begin with the `expose` method. SimpleMenu's `expose` method does no drawing of its own. It simply calls the `expose` methods of the gadget children. However, it compares the region passed into its `expose` method to determine which gadgets need redrawing. Example 12-11 shows SimpleMenu's `expose` method.

Example 12-11. SimpleMenu: expose method calling gadget children's expose methods

```
#define ForAllChildren(smw, childP) \
    for ( (childP) = (SmeObject *) (smw)->composite.children ; \
        (childP) < (SmeObject *) ( (smw)->composite.children + \
        (smw)->composite.num_children );  (childP)++ )

/* ARGSUSED */
static void
Redisplay(w, event, region)
Widget w;
XEvent * event;
Region region;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject * entry;
    SmeObjectClass class;

    if (region == NULL)
        XClearWindow(XtDisplay(w), XtWindow(w));

    /*
     * Check and Paint each of the entries - including the label.
     */

    ForAllChildren(smw, entry) {
        if (!XtIsManaged ( (Widget) *entry))
            continue;

        if (region != NULL)
            switch(XRectInRegion(region, (int) (*entry)->rectangle.x,
                    (int) (*entry)->rectangle.y,
                    (unsigned int) (*entry)->rectangle.width,
                    (unsigned int) (*entry)->rectangle.height)) {
                case RectangleIn:
                case RectanglePart:
                    break;
                default:
                    continue;
                }
        class = (SmeObjectClass) (*entry)->object.widget_class;

        if (class->rect_class.expose != NULL)
            (class->rect_class.expose)((Widget) *entry, NULL, NULL);
    }
}
```

Note that this `expose` method is also called from elsewhere in the widget code (specifically, from the `resize` and `geometry_manager` methods) to redraw the gadgets. In these cases, the region passed in is set to `NULL`, and the method clears its window and redraws all the gadgets.

Also note how this `expose` method invokes the `expose` methods of its children. All `expose` methods (and in fact all methods) are stored in the class structure, not the instance structure. Composite widgets keep only a list of the instance structures of their children. However, one field in each instance structure points to the class structure for that child. This

is the `widget_class` field of the Object instance part.* In this example, the *`entry`* counter variable is a pointer to the gadget ID (opaque pointer to the instance structure) of one of the children. Another variable, *`class`*, declared as a pointer to the `SmeObjectClass` class structure (the expected class of the children), is set to the `widget_class` field in the instance structure of one of the children. Then the `expose` field of this class structure is checked to see if it is `NULL`, and if not it is invoked.

Note that the class of the children is hardcoded in this method. This widget can manage only Sme widgets and its subclasses.

The `resize` method of SimpleMenu must resize the children when it is resized itself. (Actually, this is unlikely, since the SimpleMenu widget itself is a subclass of Shell and is therefore not managed by any parent.) This method is invoked only when the user resizes the menu using the window manager. Since this widget has the authority to determine the geometry of its children, it can simply resize them. This particular resize method (shown in Example 12-12) simply sets their width to be the same as its own.

Example 12-12. SimpleMenu: resize method

```
static void
Resize(w)
Widget w;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject * entry;

    if ( !XtIsRealized(w) ) return;

    ForAllChildren(smw, entry)  /* reset width of all entries. */
        if (XtIsManaged( (Widget) *entry))
            (*entry)->rectangle.width = smw->core.width;

    Redisplay(w, (XEvent *) NULL, (Region) NULL);
}
```

Notice that this `resize` method invokes the `expose` method (`Redisplay`) because the gadgets don't have `resize` methods, and will not redraw themselves in response to their size change.†

Now let's look at SimpleMenu's actions. Their only purpose is to call the gadgets' actions when the appropriate events arrive. These actions are added in the usual way: they are declared at the top of the *.c* file, then registered with an action list that is entered into the class structure initialization, and then defined. One of the three actions is shown in Example 12-13.

*The Core instance part structure (not complete) is the concatenation of the instance parts of the three superclasses Object, RectObj, and the unnamed class. Therefore, it also includes a `widget_class` field. Since composite widgets do not normally need to invoke the methods of their children, you shouldn't need to access this field.

†The gadget children could have `resize` methods, and this resize method could call the children's `resize` methods. The gadget's `resize` methods would simply call their `expose` method. However, this does exactly the same thing as the code shown while being more complicated.

Example 12-13. SimpleMenu: the Notify action routine

```
/* ARGSUSED */
static void
Notify(w, event, params, num_params)
Widget w;
XEvent * event;
String * params;
Cardinal * num_params;
{
    SimpleMenuWidget smw = (SimpleMenuWidget) w;
    SmeObject entry = smw->simple_menu.entry_set;
    SmeObjectClass class;

    if ( (entry == NULL) || !XtIsSensitive((Widget) entry) )
        return;

    class = (SmeObjectClass) entry->object.widget_class;
    (class->sme_class.notify)( (Widget) entry );
}
```

This action determines whether the chosen entry is sensitive and, if so, calls the `notify` method of that gadget. As described above in the section on the gadget children, gadgets define their actions as methods so that they can conveniently be called by their parent. Since these methods are stored in the class structure not the instance structure, this is done using the technique described above for the `expose` method.

Although not critical to its handling of gadgets, SimpleMenu does one more interesting thing. It registers the `PositionMenuAction` action in the global application action list (as opposed to the internal widget action list) so that the application or application-defaults file can refer to this action in translation tables without needing to register the action. This action can be triggered by any type of event in the widget and positions the menu according to data in the event type. (SimpleMenu has a resource that controls whether this placement process makes sure that the menu is not off the screen.)

A widget can add an action to the global action list by calling `XtAddAction` just like an application would, but from its `class_initialize` method.

Any composite widget that is capable of managing gadgets must declare a Composite extension structure in the *.c* file and set the `accepts_objects` field of that structure to `True`. It must then set the pointer to the extension structure into the Composite class part structure in the `class_part_initialize` method. Extension structures were introduced in Chapter 5, *Inside a Widget*, and are discussed further in Chapter 13, *Miscellaneous Toolkit Programming Techniques*. Example 12-14 shows this code from *SimpleMenu.c*.

Example 12-14. SimpleMenu.c: Setting accepts_objects in the Composite extension structure

```
CompositeClassExtensionRec extension_rec = {
    /* next_extension */  NULL,
    /* record_type */     NULLQUARK,
    /* version */         XtCompositeExtensionVersion,
    /* record_size */     sizeof(CompositeClassExtensionRec),
    /* accepts_objects */ TRUE,
};

static void
```

Example 12-14. SimpleMenu.c: Setting accepts_objects in the Composite extension structure (continued)

```
ClassPartInitialize(wc)
WidgetClass wc;
{
    SimpleMenuWidgetClass smwc = (SimpleMenuWidgetClass) wc;

/*
 * Make sure that our subclass gets the extension rec too.
 */

    extension_rec.next_extension = smwc->composite_class.extension;
    smwc->composite_class.extension = (XtPointer) &extension_rec;
}
```

This code, with names changed, will appear in all gadget parents.

13

Miscellaneous Toolkit Programming Techniques

This chapter describes various Xt functions that have not been treated elsewhere in the book.

In This Chapter:

Errors and Warnings 399
Objects 402
Macros For Getting Information 403
The Keyboard Focus and accept_focus Method 404
Keyboard Interpretation 406
Memory Allocation 408
Action Hooks and Calling Actions Directly 408
Xt Grabbing Functions 409
File Finding and Internationalization 409
Application Contexts 410
  Multiple Application Contexts 411
  Rewriting XtAppMainLoop for Multiple Application Contexts 412
  Functions Used with Multiple Application Contexts 412
Multiple Top-level Shells 413
Connecting to Multiple Servers 413
Class Extension Structures 414

13
Miscellaneous Toolkit Programming Techniques

This chapter discusses various Xt facilities that didn't fit neatly into any other chapter. Many of them are functions provided mostly because Xt uses them internally, and they are unlikely to be useful in application or widget code. Some of them are quite important for accomplishing certain tasks. You should scan the contents of this chapter to familiarize yourself with these facilities so that you will be aware of them.

The topics covered are errors and warning messages, objects, a description of all of Xt's macros and functions for getting information, the Core `accept_focus` method, how to interpret key events, Xt's facilities for memory management, making global grabs, file finding and internationalization, multiple application contexts, multiple top-level shells, and connecting to multiple servers.

13.1 Errors and Warnings

There are several broad categories of errors that may occur in Xt applications. One is the X server error, which is a form of event that tells the client that some parameter in an earlier request was illegal, or that no more server memory is available. A second is the connection failure error generated by Xlib when the connection with the server fails (usually due to a system crash or network interruption). Xlib provides the `XSetErrorHandler` and `XSetIOErrorHandler` functions to allow the application to provide a routine to handle these two types of errors. Xt provides no interface to these routines—Toolkit applications must use the Xlib routines to customize these error handlers (Xlib uses default error handlers when the application does not use these routines to specify them). For a description of these error handlers and the routines for changing them, see Volume One, *Xlib Programming Manual*.

A third category is made up of error and warning messages that Xt reports when function parameters are specified improperly, when a translation is incorrectly specified, and for many other reasons. For a complete listing of all errors and warnings that can be generated by Xt, see Volume Five, *X Toolkit Intrinsics Reference Manual*, Appendix D, *Standard Errors and Warnings*. Xt provides separate parallel routines for errors and for warnings. The difference between Xt errors and Xt warnings is that errors are fatal and the application exits after print-

ing the error, while warnings are nonfatal and the application continues. The main purpose of these facilities is to generate consistent messages.

Two levels of interface are provided:

- A high-level interface that takes an error name and class and looks the error up in an error resource database. The high-level fatal error handler is invoked by a call to `XtAppErrorMsg`; the high-level nonfatal error handler is invoked by a call to `XtAppWarningMsg`. The high-level functions construct a string and pass it to the lower-level interface.
- A low-level interface that takes a simple string, which is printed out as the error message. The low-level fatal error handler is invoked by a call to `XtAppError`; the low-level nonfatal error handler is invoked by a call to `XtAppWarning`.

These error-reporting interfaces are used internally by Xt, but widget or application code can also use them. For example, when a resource is given an illegal value in a resource file, the widget or application can report the error or warning to the user (which depends on whether the widget or application can continue after the error—most widgets issue only warnings and then fall back on their default value).

The low-level handlers are much easier to use, but they do not support internationalization (alternate languages) at all since the messages are hardcoded in the application. The high-level handlers can potentially support internationalization, but not as elegantly as the normal resource database since Xt searches for the error database file in a fixed location, not using the language string. If you want your application or widget to run in more than one language, you should use the high-level handlers, but only one database for one language can be installed on a system at a time. This is likely to be improved in later releases of Xt.

To use the low-level handlers, you specify the string message as the sole argument to `XtAppError` or `XtAppWarning`.

Contrary to what you might expect, the high-level handlers `XtAppErrorMsg` and `XtAppWarningMsg` are actually harder to use than the low-level handlers. You must pass six arguments to the calls that generate the errors or warnings, and then to take advantage of their benefits you must set up an error resource database. The first three arguments are the name, type, and class of the error. The use of these three arguments is not yet standardized since they are not widely used. However, in Xt itself, the name identifies the error message, and the type identifies the task that was in progress when the error occurred (or the section of code). The class, within Xt, is always `XtToolkitError`. The three remaining arguments of `XtAppErrorMsg` and `XtAppWarningMsg` are a default message, a parameter list, and the number of parameters. The default message will be printed only if no matching message is found in the database. Because Xt does not define or install any error database, it uses these default messages only, and ignores the name, type, and class information. (Xt uses the high-level handlers so that an error resource file can be installed to print all the errors in a foreign language.) The parameter list is used together with the message in the database. The message may be in standard *printf* format, and the parameters are used to fill in any variable fields.

Example 13-1 shows one of the rare cases where XtAppErrorMsg is invoked in the Athena widgets.

Example 13-1. How to invoke XtAppErrorMsg (from AsciiSrc.c)

```
if (src->ascii_src.string == NULL)
    XtAppErrorMsg(XtWidgetToApplicationContext(src),
            "NoFile", "asciiSourceCreate", "XawError",
            "Creating a read only disk widget and no file \
            specified.", NULL, 0);
```

The error resource database is stored in a file, */usr/lib/X11/XtErrorDB*, under most UNIX-based operating systems. Since this database is made up of one file, you must append the resource settings you need to this file rather than replacing it. The resource name searched for in the database is the concatenation of the *name* and *type* arguments specified in the calls to XtAppErrorMsg or XtAppWarningMsg.

You can redefine the routine that prints the message in order to change the fixed part of the message or to add features like logging of errors and warnings. Use XtAppSetErrorMsgHandler and XtAppSetWarningMsgHandler (if you are using the high-level handlers) or XtAppSetErrorHandler and XtAppSetWarningHandler (if you are using the low-level handlers). See the reference pages for XtErrorMsgHandler(2) and XtErrorHandler(2) in Volume Five, *X Toolkit Intrinsics Reference Manual*, for a description of how to define a new error or warning handler. The default error and warning messages printed are:

```
X Toolkit Error:   message.          (for errors)
X Toolkit Warning: message.          (for warnings)
```

Remember that Xt itself uses these messages (not just your widget code), so that they must remain appropriate when called from anywhere in the Xt, widget, or application code. If you want the message to identify the name of the widget set or widget, you must include this information in the part of the message filled in from the string you pass or from the resource database.

Table 13-1 summarizes Xt's calls for issuing errors and warnings and for modifying the messages issued.

Table 13-1. Xt Error and Warning Message Utilities

Message	Low Level	High Level
Issue Error	XtAppError	XtAppErrorMsg
Issue Warning	XtAppWarning	XtAppWarningMsg
Set Error Handler	XtAppSetErrorHandler	XtAppSetErrorMsgHandler
Set Warning Handler	XtAppSetWarningHandler	XtAppSetWarningMsgHandler

Note, however, that for the high-level routines that use the error and warning resource database, there is only one database common to all application contexts, at least in the sample implementation of Xt provided by MIT under R3 and R4.

When writing a high-level error or warning handler you will need to call `XtGetErrorDatabase` to get a pointer to the error resource database and `XtGetErrorDatabaseText` to get the message for a particular set of arguments passed to `XtAppErrorMsg` or `XtAppWarningMsg`. For details on how to use these functions, see the reference pages in Volume Five, *X Toolkit Intrinsics Reference Manual*.

`XtDisplayStringConversionWarning` is a convenience routine to be used in resource type converters that convert from `XtRString` to any representation type. It calls `XtAppWarningMsg` with the appropriate arguments to issue a suitable warning. Note however, that the class used is `XtToolkitError`. It may be better to use a class that describes the widget or widget set that defines the converter.

13.2 Objects

Most of this book describes how to use widgets, which are subclasses of the Core widget class. Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*, also discusses using gadgets, which are subclasses of the RectObj widget class. As described there, gadgets are windowless widgets that are especially useful for menu panes, because they cut back on the overhead that would be required to implement the same thing using a widget for each pane.

The third and final Xt class you can subclass is the Object widget class. Objects are windowless widgets like gadgets, but they lack geometry resources, sensitivity resources, and the `expose` method. They provide support for resources and callbacks, and that is all.

The primary use of objects is to create replaceable subparts of a widget. For example, the R4 Athena Text widget uses objects to implement its source and sink, which control the storage and the display of the data respectively. By replacing the source and sink, you could develop a multi-color or multi-font editor without having to rewrite the central widget which implements all the editing commands.

An object has the following methods: `class_initialize`, `class_part_initialize`, `set_values`, and `get_values_hook`. All these methods have the same purposes as in widgets.

While a gadget uses part of its parent's window, an object shares its parent's entire window, and is not managed at all by its parent since it has no geometry. The parent of an object is normally a widget that is a subclass of Core but not Composite or Constraint (unlike gadget parents).

But like gadgets, objects depend on cooperation by their parent, and require a specially designed parent. The object parent's methods make calls to functions defined by the object child instead of doing the work themselves. The object child's functions can be either semi-public functions or they can be class methods defined by the object. The latter is done so that subclasses of the object can replace the functions.

As an example, let's look at how the Athena Text widget handles drawing. The drawing is done by the TextSink object, and its subclass AsciiSink. Although an object has no `expose` method, it provides an equivalent function that draws on the parent. This function is either a semi-public function (public to its parent but not to the application writer), or it is a class method. The Text widget uses both techniques. TextSink provides a semi-public function `XawTextSinkDisplayText`, which is called by Text whenever drawing is needed.

A object `XawTextSinkDisplayText`, this function calls one of TextSink's class methods called DisplayText. DisplayText is a class method so that subclasses of TextSink can replace it or inherit it. AsciiSink does replace this method with code that draws the text in a single constant-width font, in the foreground and background colors. So in order to write a Text widget that draws in more than two colors, you would just have to write an object that is a new subclass of TextSink.

When Text calls `XawTextSinkDisplayText`, this function calls one of TextSink's class methods called DisplayText. DisplayText is a class method so that subclasses of TextSink can replace it or inherit it. AsciiSink does replace this method with code that draws the text in a single constant-width font, in the foreground and background colors. So in order to write a Text widget that draws in more than two colors, you would just have to write an object that is a new subclass of TextSink.

An object may assume that it should draw on its parent, but this is not necessarily the case, since an object could have an object as its parent. An object also has no idea of its own size, since it has no geometry data in its instance structure, and therefore doesn't by itself know what size window it is drawing into. To solve both these problems, `XawTextSink-DisplayText` passes the parent's widget ID and the parent's size into the objects drawing code.

13.3 Macros For Getting Information

Xt provides several macros and functions for getting information about widgets. Some of these, such as `XtIsRealized` and `XtIsManaged`, you have seen before in the context of widget methods.

Some of these are macros and some are functions, and some are macros when used in widget code and functions when used in application code. This does not affect how they can be used, so we won't bother to specify which can be both functions and macros. We will use the term "macro" for all of these informational routines. In Volume Five, they are listed alphabetically, together with all of the Intrinsics functions.

Xt provides two basic macros for determining the class of a widget: `XtIsComposite` and `XtIsSubclass`. These are primarily used internally by Xt to implement geometry management, but you may find a use for them. For example, you might write a composite widget that uses `XtIsComposite` to treat composite children differently than simple children, or uses `XtIsSubclass` to treat constraint children or one of your own classes uniquely. There are lots of convenience functions for `XtIsSubclass` that determine if a widget is a subclass of a particular class. These are `XtIsObject`, `XtIsRectObj`, `XtIsWidget`, `XtIsComposite`, `XtIsConstraint`, `XtIsShell`, `XtIsOverrideShell`, `Xt-`

`XtIsWMShell`, `XtIsVendorShell`, `XtIsTransientShell`, `XtIsTopLevelShell`, and `XtIsApplicationShell`.

You have already seen `XtIsManaged` used in composite widgets. See Chapter 11, *Geometry Management*.

You have also already seen `XtIsRealized` used in various methods to make sure a widget has a window before operations are attempted on the window. For example, the `expose` method calls `XtIsRealized` before drawing into the window.

`XtIsSensitive` checks the value of the `XtNsensitive` resource for a widget and its ancestors. If any of them is `FALSE`, it returns `FALSE`. Remember that sensitivity controls whether a widget responds to user events.

`XtHasCallbacks` lets you tell whether a widget class has a callback of a certain resource name, and whether any callback functions have actually been added to it. It returns the `enum` value `XtCallbackNoList` if there is no callback list, `XtCallbackHasNone` if there is a callback list with no functions on it, and `XtCallbackHasSome` if there is a callback list containing functions pointers.

`XtNameToWidget` searches a hierarchy for the widget ID of the specified widget instance name. Its primary use from the application is to get the IDs of the child widgets of a compound widget such as Dialog, so that their resources can be set directly. This is a violation of the rules of data hiding, however, and is not recommended. In widget code, XtNameToWidget is used to provide a layer of abstraction so that widgets can be identified using string names. For example, it is used by the converter defined by Form that allows widget names to be specified in resource files. The opposite function, which returns a widget instance name given the widget ID, is `XtName` (which perhaps should have been called "XtWidgetToName").

`XtWindowToWidget` gives you the `Widget` which corresponds to the specified X window ID. This is used mainly by Xt, but you may find a use for it.

`XtDisplayOfObject`, `XtScreenOfObject`, and `XtWindowOfObject` search the parental hierarchy of an object to discover the closest windowed ancestor and then return a pointer to a `Display` structure, a pointer to a `Screen` structure, or a `Window` ID. These macros are useful for making Xlib calls from within code that implements a subclass of Object.

`XtGetApplicationNameAndClass` returns the name and class strings of an application. The name is usually `argv[0]` stripped of any directories, while the class is the string passed as the second argument of `XtAppInitialize`. These are the name and class used by Xt to look up resources for the application and its widgets. You are unlikely to need this function.

13.4 The Keyboard Focus and accept_focus Method

The keyboard focus is the window to which the server sends keyboard events. The window manager controls which window this is. Under click-to-type window managers, the window that was most recently clicked on (usually with some keyboard key held) receives the keyboard focus. Under a real-estate-driven or pointer-following window manager, the keyboard focus is always the root window, which results in keyboard events being sent to the application (and to the individual window) the pointer is currently in. In any case, even click-to-type window managers set the keyboard focus only to children of the root window—the application top-level windows.

The Core class part structure includes a field for the `accept_focus` method. This method lets a widget set the keyboard focus to one of its children when it gets the keyboard focus. A typical example is an application that wants to set the keyboard focus to the text entry child of a dialog box whenever the dialog box is given the keyboard focus by the window manager. This would be done so that the user can type with the pointer anywhere in the dialog widget instead of just with the pointer in the text entry widget.

To implement this example, the text entry child would need an `accept_focus` method that would set the keyboard focus to itself using the Xlib call `XSetInputFocus`, and the dialog box would need an `accept_focus` method that called `XtCallAcceptFocus` on the text entry widget child. The application can call `XtCallAcceptFocus` on the dialog widget in response to `FocusIn` events to start this process, and set the focus back to `PointerRoot` on `FocusOut` events.* For details on these events and how to set the keyboard focus with `XSetInputFocus`, see Volume Two, *Xlib Reference Manual*. (The Athena Dialog widget and Text widget do not define the appropriate `accept_focus` methods for this process of settting the keyboard focus to work.) This procedure is illustrated in Figure 13-1.

The `accept_focus` returns a Boolean value to report whether it succeeded in setting the keyboard focus, and `XtCallAcceptFocus` returns this same value.

The `XtSetKeyboardFocus` function can be used to redirect keyboard events that occur anywhere within a dialog box to a child of the dialog, usually a text entry widget. (This function affects only Xt's event dispatching, and is independent of the X server keyboard focus, which has a similar but more widespread effect.) The Dialog widget can itself call this function instead of calling `XtCallAcceptFocus`. Dialog can also provide a resource or public function to allow the application to control it.

*`XtCallAcceptFocus` is not provided in MIT's version of the R3 Intrinsics. It may be present in some vendor's versions of R3, but this is unlikely. It is available in all versions of the R4 Intrinsics.

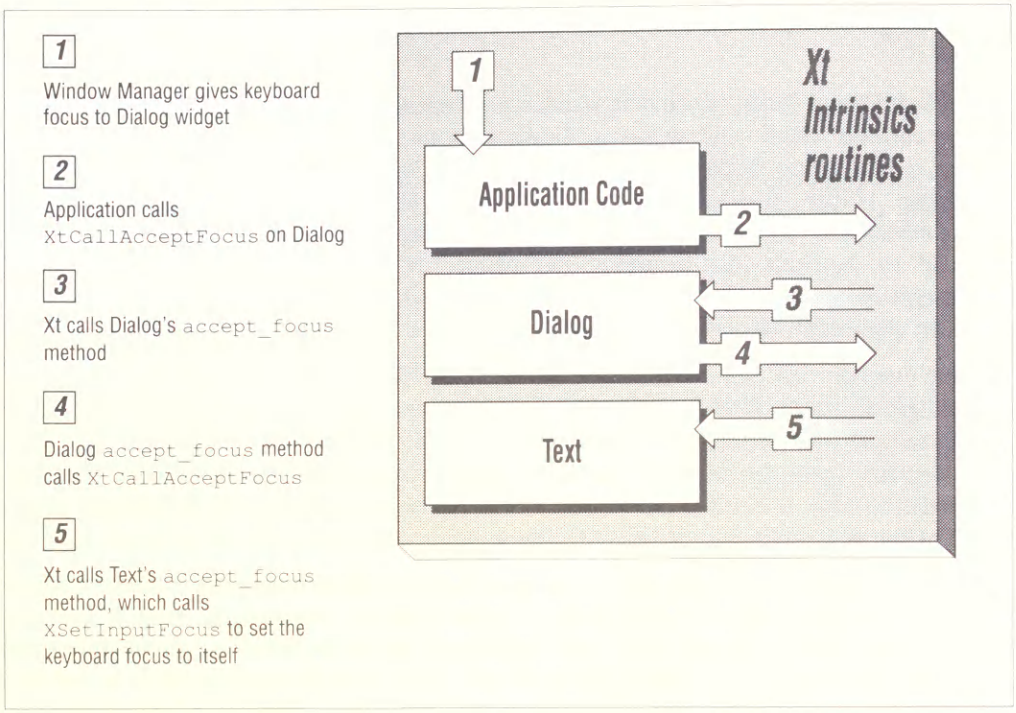


Figure 13-1. Using XtCallAcceptFocus to set the keyboard focus to a child

This procedure for giving the child of Dialog the keyboard focus is necessary because the application can't find out the name of the child of Dialog that should have the focus without breaking widget encapsulation rules.

13.5 Keyboard Interpretation

Keyboard handling in X is designed so that you can write a program that will operate on systems with widely different physical keyboards. To accomplish this, there are several layers of mappings:

- The first mapping is between physical keys and keycodes (a number for each key), and varies between servers. A `KeyPress` event includes only the keycode and information about what other keys and buttons were being held at the time of the keypress. Programs that interpret keycodes directly will operate on only one type of system.
- The next mapping is between keycodes and keysyms, which are symbolic constants beginning with `XK_` that represent the meaning of a key press. This mapping takes into account whether Shift or other modifier keys were being held during the press. Xlib provides the routine `XLookupString` that converts the keycode in a key event to the appropriate keysym. Portable programs use keysyms to interpret key events. The keycode to keysym mapping is server-wide. It can be changed, but this is normally done

only to accomplish radical changes in the placement of keys such as changing a QWERTY style keyboard to DVORAK.

- The final mapping is between keysyms and strings. For printing characters, `XLookupString` also returns a string representation of the interpretation of the key pressed. For example, if the key marked *A* was pressed with no other keys held down, the string returned would be *a*. A text entry widget, for example, would append this string to the string being displayed, but modify the string in other ways to handle keysyms that do not have a string representation such as `XK_Backspace`. The values of keysyms are arranged logically so that all printing characters have a particular range.

When you write an action that accepts key events, you will usually need to interpret the meaning of the key pressed. Xt provides its own interface to `XLookupString`: `XtTranslateKeycode`. You pass several fields of the key event to `XtTranslateKeycode`, and it returns the keysym. However, `XtTranslateKeycode` does not return the string interpretation of the key event that would be returned by `XLookupString`. If you need that string, you will have to call `XLookupString`.

Xt provides `XtTranslateKeycode` because Xt also provides routines for changing the way the translation returned by `XtTranslateKeycode` is done. `XtSetKeyTranslator` allows you to specify your own procedure to convert from the key event information to a keysym. The default key event translation procedure is `XtTranslateKey`, and so you can restore the default translator if necessary, and so that you can call it from your own translator to get default translations (you need to add only the code that makes the translations not done by the default translator). See `XtKeyProc` in Volume Five, *X Toolkit Intrinsics Reference Manual*, for details on providing a key event translation procedure.

Among these routines for modifying the interpretation of key events is a facility for changing the handling of capitalization. For example, most keyboards have the question mark (*?*) symbol over the slash (/) symbol on one key. The standard case converter converts a press of this key with the Shift key held down to the `XK_question` keysym. In rare cases a keyboard may have a different symbol over / and put *?* somewhere else. Also, some keyboards have two or more symbols on a single key, some of which are not represented at all by standard keysyms. The case converter handles these situations. The case converter is usually called from the key translator described above. To call the case converter, use `XtConvertCase`, and to change the case converter, call `XtRegisterCaseConverter`. See `XtCaseProc` in Volume Five, *X Toolkit Intrinsics Reference Manual*, for details on writing a case converter procedure.

Note that the translation manager uses these same key translation and case converter routines to interpret translation tables. Therefore, make sure that you add features only to them, keeping existing features.

Xt also provides two routines that may help in interpreting key events: `XtGetKeysymTable` and `XtKeysymToKeycodeList`. The former returns the entire mapping of keycodes to keysyms for the server, while the latter tells you what keycodes are listed for the specified keysym. Neither function is necessary for routine keyboard handling.

From within an action routine, you can call XtGetActionKeysym to get the keysym that resulted in the action being called. This can be very useful, since the event passed to the action contains only the keycode of the key that was pressed. However, there is another way to acheive a similar result. You can use string parameters of actions to pass in the keysym that you specified in the translation table. For example, if you provide the translation `:<Key>q : Quit(q)`, the Quit action will be passed the string "q" in its *params* argument. However, XtGetActionKeysym is very useful if you translate a wide range of key events to one action, and then want to distinguish between them in the action.

13.6 Memory Allocation

Xt provides routines for performing routine memory allocation and deallocation. The routines XtMalloc, XtCalloc, XtRealloc, and XtFree are equivalents of the standard C routines malloc, calloc, realloc, and free but they add error checking and reporting. The allocation routines make sure the allocation succeeded, and if it did not, they print a (fatal) error message. XtFree makes sure the passed pointer is not NULL before calling free.

XtNew is a macro which allocates storage for one instance of the passed type and returns a pointer. For example, XtNew(XtCallbackList) allocates storage for one callback list structure. XtNewString is a macro that allocates storage for a string, copies the string into the new storage, and returns the pointer. For example, a string can be copied into new storage using the following:

```
static String buf[] = "How do you do?";
String p;

p = XtNewString(buf);
```

After this sequence, p points to a separate string that contains "How do you do?" Then buf can be changed without affecting p.

13.7 Action Hooks and Calling Actions Directly

Xt allows you to register any number of functions to be called whenever any action in an application context is invoked. This is done with XtAppAddActionHook. Note that there is just one "action hook" in the application context, so that all the action hook functions registered for that application context are called whenever *any* of the actions in that application context are invoked. The registration does not specify any particular action or any particular widget.

The main reason for registering an action hook is to record all the actions that were invoked in an application, so that they can be played back later using XtCallActionProc. An action hook function is called with all the same arguments that are passed to an action, plus the string name of the action. The action hook function would store this information as a unit

each time it was called. When it comes time to play back the recorded actions, it would pass all the information in each unit to `XtCallActionProc`.

An action hook can be removed with `XtRemoveActionHook`.

13.8 Xt Grabbing Functions

The most common use of grabs in an application is for pop up menus and dialog boxes, as described in Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*. As described there, Xt has its own *grab mode* that controls the distribution of events within one application, which can be used to restrict events to one pop up in a cascade or allow events to go to any pop up in a cascade. As also described there, pop-up menus in particular need a passive global grab of the pointer in order to detect button releases that occur completely outside the application so that menus can be popped down properly. A passive global grab actually instructs the server to redirect events to a certain window. All the grabbing needs of pop ups are satisfied by the built-in action `XtMenuPopup`, or by the function `XtPopupSpringLoaded`, which can be used in a callback function. If necessary, you can write your own version of `XtMenuPopup` and register it with `XtRegisterGrabAction` so that the appropriate passive global grab is in effect.

The above facilities should be quite sufficient for your needs in the area of pop ups. However, it is possible that you may need an global grab for some other purpose. Perhaps you need all keyboard events in one window for a short time and don't want to change existing translations or accelerators for keyboard events in other widgets. This is only one plausible scenario.

The Xlib functions for grabbing are `XGrabKey`, `XGrabButton`, `XGrabKeyboard`, and `XGrabPointer`. Each of these functions has an analogue for ungrabbing. But Xt provides its own version of all eight of these functions. You should use the Xt versions because they are integrated into Xt and take care of things like making sure the widget that is to get the grab has been realized. In other words, if you do need the server to redirect events for you using a grab, use the following functions: `XtGrabKey`, `XtGrabButton`, `XtGrabKeyboard`, `XtUngrabKeyboard`, and `XtUngrabPointer`. For more on global grabs, see Chapter 8 in Volume One, *Xlib Programming Manual*.

13.9 File Finding and Internationalization

Xt's facilities for allowing an application to run in different languages are mostly built into the resource database mechanism. Where Xt searches for resource files depends on a language string, which can be set within resources at runtime or with an environment variable to affect one user's environment. This was described in Chapter 9, *Resource Management and Type Conversion*.

The functions that Xt uses to implement its search for resource files are also available to you for searching for application files. This allows you to provide separate data files for each language for a help system, for example. The primary function you would use to do this is

`XtResolvePathname`, since it searches directories in a standard order based on X/Open Portability Guide conventions.

If you want to find a file but are not interested in internationalization, you can use `XtFindFile`.

13.10 Application Contexts

The introduction to application contexts in Section 3.8 described their use in 99 percent of applications. As you may recall, their purpose is chiefly to attain portability to certain systems that do not provide a separate address space for each process.

Xt provides parallel versions of many routines—one set that uses the default application context, and one set that has an explicit application context argument. The routines that use the default application context are remnants of an earlier release when application contexts did not work properly. To acheive the desired portability, you must now use the versions with the explicit argument. We have done this throughout this book. In Volume Five, *X Toolkit Intrinsics Reference Manual*, the reference pages for all the functions that use the default application context note the fact that they should no longer be used.

Table 13-2 shows the complete list of routines that have two versions.

Table 13-2. Xt Routines That Use Default and Explicit Application Contexts

Default	Explicit
(registering functions)	
`XtAddActions`	`XtAppAddActions`
`XtAddConverter`	`XtAppAddConverter`
`XtAddInput`	`XtAppAddInput`
`XtTimeOut`	`XtAppTimeOut`
`XtWorkProc`	`XtAppWorkProc`
(creating shells)	
`XtCreateApplicationShell`	`XtAppCreateShell`
(event dispatching)	
`XtMainLoop`	`XtAppMainLoop`
`XtNextEvent`	`XtAppNextEvent`
`XtPeekEvent`	`XtAppPeekEvent`
`XtPending`	`XtAppPending`
`XtProcessEvent`	`XtAppProcessEvent`
(error and warning messages)	
`XtError`	`XtAppError`
`XtErrorMsg`	`XtAppErrorMsg`
`XtGetErrorDatabase`	`XtAppGetErrorDatabase`
`XtGetErrorDatabaseText`	`XtAppGetErrorDatabaseText`
`XtSetErrorHandler`	`XtAppSetErrorHandler`
`XtSetErrorMsgHandler`	`XtAppSetErrorMsgHandler`
`XtSetWarningHandler`	`XtAppSetWarningHandler`

Table 13-2. Xt Routines That Use Default and Explicit Application Contexts (continued)

Default	Explicit
XtSetWarningMsgHandler	XtAppSetWarningMsgHandler
XtWarning	XtAppWarning
XtWarningMsg	XtAppWarningMsg
(selection timeouts)	
XtGetSelectionTimeout	XtAppGetSelectionTimeout
XtSetSelectionTimeout	XtAppSetSelectionTimeout
(action hooks)	
(no equiv)	XtAppAddActionHook
(toolkit initialization)	
(no equiv)	XtAppInitialize
(resources)	
(no equiv)	XtAppSetFallbackResources
XtSetTypeConverter	XtAppSetTypeConverter
(no equiv)	XtAppReleaseCacheRefs

Note that XtCreateApplicationShell and XtAppCreateShell have names that don't follow the example set by all the rest.

13.10.1 Multiple Application Contexts

The use of more than one application context in a single program presents possibilities that you might wish to explore. Having more than one application context in the same program allows you to have one program that when run looks like two or more independent programs. This approach saves disk space and memory on systems that don't provide shared libraries, since the grouped programs can share a single copy of the libraries.*

Having two application contexts makes each sub-application more separate than if they were just under different top-level Shell widgets. Each widget class has an action list, and each application context has a separate context-wide action list. When the translation manager looks for an action, it looks in the widget class action list first, and then the application context action list. Therefore, each sub-application could add an action to its application context without conflict with another sub-application adding a different action of the same name.

On parallel processing machines, each separate application context could run in parallel. However, it is difficult to write portable code to take advantage of this, since each architecture has different conventions for indicating parallelisms in C code.

*In SunView, many of the basic applications were grouped in a single binary probably for this reason. The Xlib and Xt libraries are quite large. For example, on a Sony NWS-841 workstation, the executable image of a "hello, world" application written with Xt uses 300K of disk space. One of the most complicated existing X applications, *xterm*, uses 450K of disk space on this system. Therefore, the various libraries account for about two-thirds of the disk space used, even for a large program.

13.10.2 Rewriting XtAppMainLoop for Multiple Application Contexts

To use multiple application contexts, you need to write your own equivalent of XtAppMainLoop to dispatch events to your multiple application contexts. This is necessary because XtAppMainLoop dispatches events only from one application context, and it never returns so you can't call it again for the other.

The available tools are XtAppNextEvent, XtAppPeekEvent, XtAppPending, XtAppProcessEvent, and the Xlib functions XFlush and XSync. Rewriting XtAppMainLoop for two or more application contexts is tricky, because you don't want to let the dispatching of any one application context get behind. It is not as simple as dispatching events from each application context alternately, since the events might not occur alternately. It is easy to get stuck waiting for events in one application context while events queue up at the other. There is little experience in how this should be done properly, and no examples in the distribution from MIT. However, hypothetically, the following describes how it could work.

To do this properly, you have to understand how Xlib's network optimization works. Xlib buffers up many types of requests and sends (flushes) them to the server as a group.* A flush is most commonly caused by a routine such as XtAppNextEvent that waits for an event if none are available. It is because XtAppNextEvent waits forever for an event that the routine could get locked waiting for events in one application context while the user types frantically in the other.

The answer is to use XtAppPending to determine whether an event is available on a particular application context, and then call XtAppProcessEvent if there is an event to process. Then continue to do the same on each other application context. However, this alone is not enough. Neither XtAppPending nor XtAppProcessEvent called in this manner cause Xlib's buffer of requests to be sent to the server. Therefore, periodic calls to XSync or XFlush are necessary to flush the output buffer. The difficult part is to call these enough to flush the buffer when necessary, but not so much as to eliminate the advantages of the buffering. There is no ideal solution to this problem.

On multi-tasking systems it is perhaps possible to fork so that each application context runs in a separate process.

13.10.3 Functions Used with Multiple Application Contexts

XtWidgetToApplicationContext and XtDisplayToApplicationContext could be useful if you use more than one application context in an application. XtWidgetToApplicationContext is also useful in widget code, to call error-issuing routines that require an application context argument, such as XtAppWarning and XtAppWarningMsg.

---

*For a detailed discussion of Xlib's network optimization and its effects, see the introduction to Volume Zero, *X Protocol Reference Manual*.

13.11 Multiple Top-level Shells

A single application can have more than one top-level window. In other words, you are not restricted to containing your application's entire user interface in a single rectangle. If you have one section of the application that is most appropriate as a long, thin vertical window that looks like a long, permanent menu, and another section that is a long, horizontal bar such as a ruler, each of these could be a separate top-level window. That way, not only is less screen space wasted than if these two windows were placed within a single rectangle, but the user can move the two windows around separately using the window manager. The user can also iconify them separately when not needed.

To create additional top-level application shell widgets, you call `XtAppCreateShell` or `XtVaAppCreateShell`. (`XtCreateApplicationShell` is now superceded.) The class specified in the call should be `topLevelShellWidgetClass`.

As you may know, a single server may have several screens attached. At present, all shells created will appear on the default screen. There is no way for the application to specify that a shell be created on a particular screen, but then again, doing this is usually unwise anyway because not many users actually have more than one screen. The user can specify which screen is considered the default screen using the *-display* command-line option.

13.12 Connecting to Multiple Servers

One of the great features of the server-client model is that a single program can connect to several servers to display on the screens of several users. For example, this would allow you to create an X-based version of the UNIX utility *wall* in which one user can make a message appear on all user's screens throughout a network. You could also create a conferencing program in which each user has a window on their screen in which they can type and view typing by others in real time, and their typing will appear on all the other user's screens.

The Xt application opens a connection with a server using `XtOpenDisplay` (this routine requires an explicit application context argument). Once you have opened the connection, you will want to create a shell widget on the server's default screen using `XtCreateApplicationShell` or `XtAppCreateShell`. Then, you create widgets for each server simply by using the appropriate Shell widget as parent. Thereafter, `XtAppMainLoop` dispatches events from all the connections to the appropriate widget.

13.13 Class Extension Structures

X Consortium standards, once adopted, can only be changed in ways that are binary and source compatible with the original standard specification. Xt became an X consortium standard in Release 3. Therefore, Release 4 and later releases are required to be source and binary compatible with Release 3. *Source compatibility* means that properly coded applications and widgets written to the R3 specification should compile and work under R4 and later releases. Source compatibility is maintained by keeping all programming interfaces intact while adding features with new interfaces. *Binary compatibility* means that widgets written and compiled with R3 must be able to be linked with R4 Xt and Xlib libraries and still run. Source compatibility is necessary but not sufficient for binary compatibility. The major addition requirement of binary compatibility is that fields added to structures must be added to the end of the structure.

In Chapter 5, *Inside a Widget*, you saw how a class structure for a widget class is built by nesting the class part structures of all its superclasses into one big structure, the class record. The X consortium could not simply add fields to the class part structures of the basic Xt classes without breaking binary compatibility with existing subclasses because the class parts of basic Xt classes appear in the middle of the class record of subclasses. Therefore, they used extension structures to add these fields.

Class extension structures allowed the X consortium to add to the class structures of basic Xt classes. The last field of each basic Xt class is called `extension`. This field can be set to a pointer to the class's extension structure, which contains its added fields. New features that required additional class structure fields were added to the Composite and Constraint classes in R4. Therefore, these two classes now have extension structures.

Some classes, such as Core, do not have an extension structure because no additional class fields have yet been necessary. They still have the `extension` field, but it is not used.

A class extension structure is defined in the private header file of a widget class. The extension structure for Composite is called `CompositeClassExtensionRec` as shown in Example 13-2.

Example 13-2. Common fields in a class extension record

```
typedef struct {
    XtPointer next_extension;/* 1st 4 mandated for all ext rec */
    XrmQuark record_type;    /* NULLQUARK; on CompositeClassPart */
    long version;            /* must be XtCompositeExtensionVersion */
    Cardinal record_size;    /* sizeof(CompositeClassExtensionRec) */
    Boolean accepts_objects;
} CompositeClassExtensionRec, *CompositeClassExtension;
```

All extension structures start with the same four fields. The only field in this structure actually used by Composite is `accepts_objects`, the use of which was described in Section 12.4.5. The `accepts_objects` field is only used by Composite widgets that have the code necessary to accept gadget children. When a widget class needs an extension feature, the widget class initializes the extension structure in its *.c* file, and then in the `class_initialize` method sets the `extension` field to point to the extension structure.

When a widget class does not need an extension feature—for example, a Composite widget that does not accept gadget children—the widget class does not provide code in the `class_initialize` method to set the `extension` field.

The *.c* file of all widget classes should initially set all `extension` fields to `NULL`. The difference between classes that use extension features and those that don't is only the presence or absence of code in `class_initialize` to set the `extension` fields.

The four required fields in an extension structure are:

next_extension	Specifies the next record in the list, or `NULL`.
record_type	Specifies the particular structure declaration to which each extension record instance conforms.
version	Specifies a symbolic constant supplied by the definer of the structure.
record_size	Specifies the total number of bytes allocated for the extension record.

When you initialize an extension structure of a given class in the Example 13-3 shows how to initialize the Composite extension structure. The first two fields are usually `NULL` and `NULLQUARK`. The reference page for each class that has an extension structure will document how to initialize the third and fourth fields.

```
CompositeClassExtensionRec extension_rec = {
    /* next_extension */  NULL,
    /* record_type */     NULLQUARK,
    /* version */         XtCompositeExtensionVersion,
    /* record_size */     sizeof(CompositeClassExtensionRec),
    /* accepts_objects */ TRUE,  /* only new field */
};
```

The `next_extension` field implies correctly that you can nest extension structures. Perhaps a later release of Xt will require the addition of more Composite class fields. These could be added to the end of the existing extension structure, or a new extension structure could be defined and a pointer to it placed in `next_extension`. This allows additions to be made to a class structure without breaking binary compatibility.

You may be wondering whether you should use extension structures when extending your own widgets. Probably not. Because you are likely to release your set of widgets as a package, you have no need for binary compatibility with previous releases (of your own). Binary compatibility would only be an advantage if you wanted to replace a few widgets from a former release and save users who may have subclassed the widgets you have replaced from recompiling their widgets. For the trouble involved, the benefit is very small.

A

OPEN LOOK and Motif

This appendix gives an overview of the widgets available in the AT&T widget set (also known as Xt+) and OSF Motif widget set. It gives a sense of the look and feel of applications developed with each set, and provides the inheritance hierarchy and overview of the available widgets.

In This Chapter:

The AT&T OPEN LOOK Widgets 423
Application Controls 424
Command Buttons 424
Exclusive and Nonexclusive Settings 426
Analog Controls 428
Composite Widgets 428
Menus and Control Areas 428
General Purpose Composite Widgets 429
Scrollbars and Scrollable Windows 429
Pop Ups 431
Text Widgets 433
Drawing Areas 433
The OSF/Motif Widgets 433
Application Controls 436
Command Buttons 436
Analog Controls 436
Composite Widgets 437
Menus and Control Areas 437
General Purpose Composite Widgets 440
Scrollable Windows 441
Pop Ups 442
Text Widgets 443
Drawing Areas 443

A
OPEN LOOK and Motif

This section provides an overview and comparison of the widgets in MIT's Athena widget set, AT&T's OPEN LOOK™ widget set (also known as Xt+), and the Open Software Foundation's Motif™.*

As we've already discussed, the Athena widgets were developed to test and demonstrate the Xt Intrinsics. They are used as the basis for some of the standard MIT clients and many public domain applications, but are not expected to be used for most commercial applications because Xaw is not a complete environment.

A number of vendors have developed proprietary widget sets. For example, Sony Microsystems offers S-windows, a widget set for its News workstations. However, given that one of the purposes of widgets is to provide a common look and feel for X applications, it is natural there should be a shakeout as vendors align themselves with one or two major contenders.

As it has turned out, the two major contenders for a graphical user-interface standard, OPEN LOOK and Motif, are put forth by the two major contenders for an underlying UNIX operating system standard, AT&T and the Open Software Foundation.

It is possible to write an application that will run under either of these widget sets, using one set of source code interspersed with `#ifdef` symbols for conditional compilation.

OPEN LOOK is somewhat unusual in that it started out not as a set of widgets, but as a user-interface specification. The specification, originally developed by Sun Microsystems with AT&T backing, was widely circulated for comment before any implementations were begun. The objective was to develop a graphical user-interface standard for UNIX workstations—one that would be implementation-independent, and, it was hoped, implemented separately by many different vendors.

At present, the two major implementations of OPEN LOOK are Sun's XView toolkit (which is not based on Xt, but instead provides an application-programmer's interface similar to Sun's proprietary SunView windowing system), and AT&T's OPEN LOOK Xt-based widget set. XView is on the R4 distribution from MIT. Both of these toolkits will be available to all AT&T UNIX System V Release 4 licensees. In our discussions, we are referring specifically to AT&T's OPEN LOOK toolkit, which does not necessarily include every OPEN LOOK

*OPEN LOOK is a registered trademark of AT&T and Motif is a registered trademark of Open Software Foundation, Inc.

feature. Nor should its implementation be considered the only way to provide features called for by OPEN LOOK.

The Open Software Foundation's Motif toolkit is based on a combination of widget sets originally developed by two OSF sponsors, Digital and Hewlett Packard. The look and feel of the widget set was proposed by HP/Microsoft. It is designed to emulate the look and feel of the IBM/Microsoft Presentation Manager standard widely expected to be adopted in the microcomputer world.

Motif's API (Application Programmer's Interface) is based on DECWindows. Digital also provided Motif with some underlying enhancements to the Xt Intrinsics (most notably an implementation of windowless widgets called *gadgets*) and various supporting utilities. Motif release 1.0, described here, is based on the R3 Intrinsics; with these enhancements. Motif release 1.1, due in late summer 1990, is based on the standard R4 Intrinsics.

Table A-1 compares the widgets available in Athena, AT&T OPEN LOOK set, and Motif.

Table A-1. Comparison of Athena, OPEN LOOK, and Motif Widgets

Simple widgets (mostly controls):

Athena	OPEN LOOK	Motif	Description
Command	OblongButton	PushButton	Invokes a command
—	—	DrawnButton	Invokes a command
Toggle	ToggleButton	RectButton	Chooses a setting
—	CheckBox*	—	Alternate way of choosing a setting
MenuButton	ButtonStack	CascadeButton	Invokes a menu, displays label
—	AbbrevStack	—	Invokes a menu, displays default
—	—	ArrowButton	Reverses direction of movement
—	ScrollingList*	List	Displays a list of selectable strings
Scrollbar	Scrollbar	ScrollBar	Scrolls through an associated window
—	Slider	Scale*	Sets (or displays) an analog value
Grip	—	—	Resize point for panes in VPaned
Label	StaticText	Label	Displays a fixed string
Text	Text	Text	Displays editable text
—	TextField	Text	Displays a single line of editable text
—	—	Separator	Displays a line or other separator

*Checkbox, ScrollingList, and Scale are technically composite widgets.

Table A-1. Comparison of Athena, OPEN LOOK, and Motif Widget (continued)

Pop ups (subclasses of shell):

Athena	OPEN LOOK	Motif	Description
SimpleMenu	Menu	MenuShell	Parents a popup menu
—	Notice	DialogShell	Displays a dialog requiring input
—	PopupWindow	—	Displays a more complex dialog
—	Help	—	Displays a help window

Composite and Constraint Widgets:

Athena	OPEN LOOK	Motif	Description
—	BulletinBoard	BulletinBoard	Free-form placement area
—	—	DrawingArea	Free-form drawing area
Box	—	—	Displays children in order added
—	ControlArea	RowColumn	Arranges children in rows or columns
Form	Form	Form	Manages children relative to each other
—	Exclusives	—	Makes RectButton children exclusive
—	Nonexclusives	—	Makes RectButton children nonexclusive
—	FooterPanel	—	Provides a consistently-sized message area
—	—	Frame	Gives consistent border to enclosed widgets
—	ScrollingList*	SelectionBox	Provides a selectable list of strings, plus a text area for entering a new value
—	—	Command	Provides a selectable list of commands
—	—	FileSelectionBox	Provides a selectable list of filenames
—	Caption	—	Displays a label and one child widget
Viewport	ScrollingWindow	ScrolledWindow	Displays a scrollable child window
—	—	MainWindow	ScrolledWindow with special appearance
VPaned	—	PanedWindow	Displays panes resizable in one direction

*Checkbox, ScrollingList, and Scale are technically composite widgets.

Comparable widgets share a line in the table. Widgets for which no equivalent occurs in a given set are indicated by a hyphen in the appropriate column. Note that comparisons are approximate only, since widgets have complex behavior that may distinguish them significantly from another widget with an ostensibly similar purpose.

The following sections provide an overview of the widgets available in the OPEN LOOK and Motif widget sets. Throughout, we contrast them with the Athena widgets, which have been used as examples in this book, to give you an idea of the additional features provided by the commercial widget sets.

Keep in mind that the look and feel of an application is controlled by the window manager as well as by the widget set. Both AT&T and OSF supply window managers to complement their widgets.

Note that both widget sets make additions to the basic X Toolkit API. In the AT&T OPEN LOOK widget set, these API additions are rather minor. There is one essential function, `OlInitialize`, which sets initial values needed by other routines and by certain widgets. `OlInitialize` creates a base window, which from the programmer's point of view is identical to the Intrinsics-supplied TopLevelShell widget class, but which automatically handles certain features of the OPEN LOOK interface. There are also several convenience functions, mostly having to do with conversions between pixel sizes and various standard units. More importantly, there is a facility for registering help screens for each element in an application.

Motif 1.0 has made more extensive API additions and even changes, modifying all of the base widget classes, and other Intrinsics features. Therefore, the Motif 1.0 Intrinsics are not compatible with the MIT R3 or R4 Intrinsics. Motif 1.0 support for windowless widgets, or "gadgets," is part of OSF's proprietary version of the Intrinsics, which is not completely compatible with the version of gadgets support in the MIT R4 release. However, Motif 1.1 is expected to be fully compatible with the MIT Release 4 Intrinsics.

In addition, Motif makes heavy use of convenience functions. Rather than using `XtCreateManagedWidget` to create each widget, there is a separate creation routine for each widget. In some cases, a convenience routine creates more than one widget. Rather than using separate calls to `XtCreatePopupShell` and `XtCreateManagedWidget` to create a pop-up shell and the dialog box it displays, you might call a function such as `XmCreateMessageDialog` to create both widgets at once. Some convenience routines create special configurations of a single, complex widget (e.g., a composite widget with specific children.)

In Motif, all resources are referred to by names beginning with `XmN` or `XmC` rather than the familiar `XtN` and `XtC`.

Motif also uses the *`call_data`* argument of callback functions extensively. Almost every widget has a structure defined as *`widgetclass`*`CallbackStruct` (e.g., `XmToggleButtonCallbackStruct`). This struct contains different fields for each widget, but each contains a field called `reason`. The `reason` field defines which callback has been called. So using this feature allows you to have a single piece of code to handle all callbacks for a widget.

In the Motif 1.0 Intrinsics, the search path for resources has been expanded to support language-specific resource defaults (to support internationalization of applications).

Both Motif and OPEN LOOK offer clear advantages over Xaw. Unfortunately, however, the choice of which one to use may depend on company marketing goals and politics rather than on clear technical merit. We encourage independent developers to try both and to base your opinions on the ease of programming and the preferences of your users, rather than on marketing hype by one side or the other. Both Motif and OPEN LOOK come with their own window manager and have defined their own protocols for communication between their Shell widgets and their window manager. By setting resources of Shell widgets you can control some aspects of how these window managers will handle your application.

A.1 The AT&T OPEN LOOK Widgets

Figure A-1 shows the overall look and feel of an OPEN LOOK application.

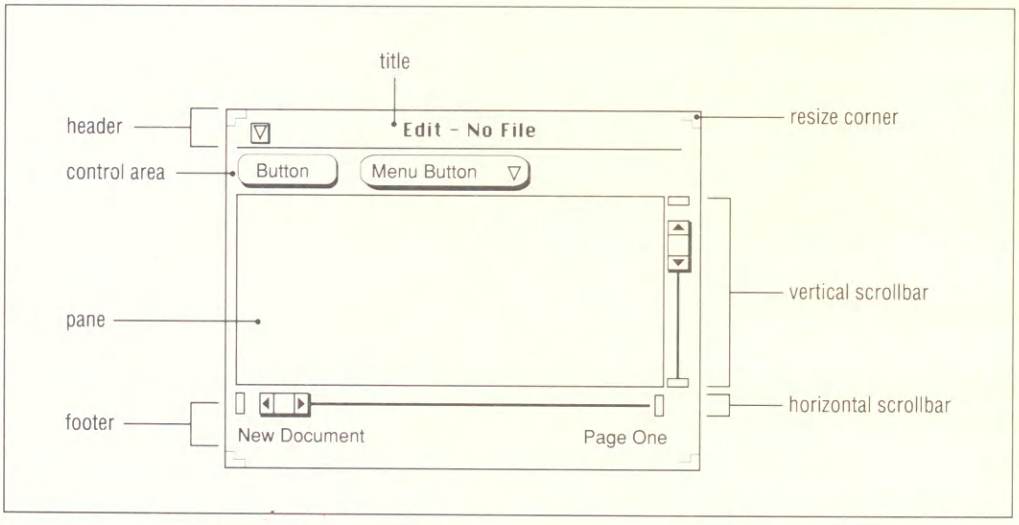


Figure A-1. An OPEN LOOK application

The base window of an application always has these elements:

- A title bar, or header, with a title supplied by the application centered in the bar. The title might be the application name, or the name of a file being edited. If the application doesn't provide a title, the title bar displays the string "Untitled."
- A "window mark" on the left side of the title bar. If the user clicks on this mark with the first (leftmost) pointer button, the window closes.
- A window menu that comes up automatically when you click on the title bar with the third (rightmost) pointer button.
- One or more panes for input and display of application data.

- One or more control areas, containing buttons that invoke application actions or menus containing additional actions. Control areas can be horizontal, vertical, or both.

Optional elements include resize corners (which allow the user to resize the application by dragging them with the first pointer button), horizontal and/or vertical scrollbars, and a footer area for displaying messages.

Some of these elements do not correspond to widgets, but are produced by the OPEN LOOK window manager, *olwm*, as window decoration. For example, the header, including title bar, window mark and window menu, are provided by the window manager, as are the optional resize corners. However, control areas, panes, scrollbars, and the footer area do correspond to particular widgets.

In addition to base windows, applications may have several kinds of pop up. Both drop-down and pure pop-up menus are supported, as well as several standard kinds of notices and dialogs. Probably OPEN LOOK's best-known feature is the "pushpin" metaphor that allows frequently-accessed pop-up menus to be kept on the screen rather than hidden again after they have been used.

The following sections discuss some of the widgets AT&T has provided to support the OPEN LOOK user interface. Figure A-2 shows the overall widget inheritance hierarchy.

Note that there are a number of widgets that are never instantiated by the application programmer, but are used by other widgets. For example, the checkbox is actually a composite widget that manages a check widget as its child! The Pushpin used in pop ups and the Magnifier used in Help windows (and the Help window itself) are examples of other widgets that are not instantiated directly.

A.1.1 Application Controls

Most applications will have at least one control area, with pointer-selectable buttons that invoke commands or menus, or choose settings.

One of the areas where OPEN LOOK clearly stands out over the Athena widgets is in the rich set of controls it provides. Athena has one kind of Command widget; OPEN LOOK has six.

A.1.1.1 Command Buttons

The Athena Command widget implements one of the most basic user-interface idioms—a button that invokes an action when you click on it with the pointer. The Athena widgets include a subclass of Command, the MenuButton widget, which includes code for placing a pop-up menu.

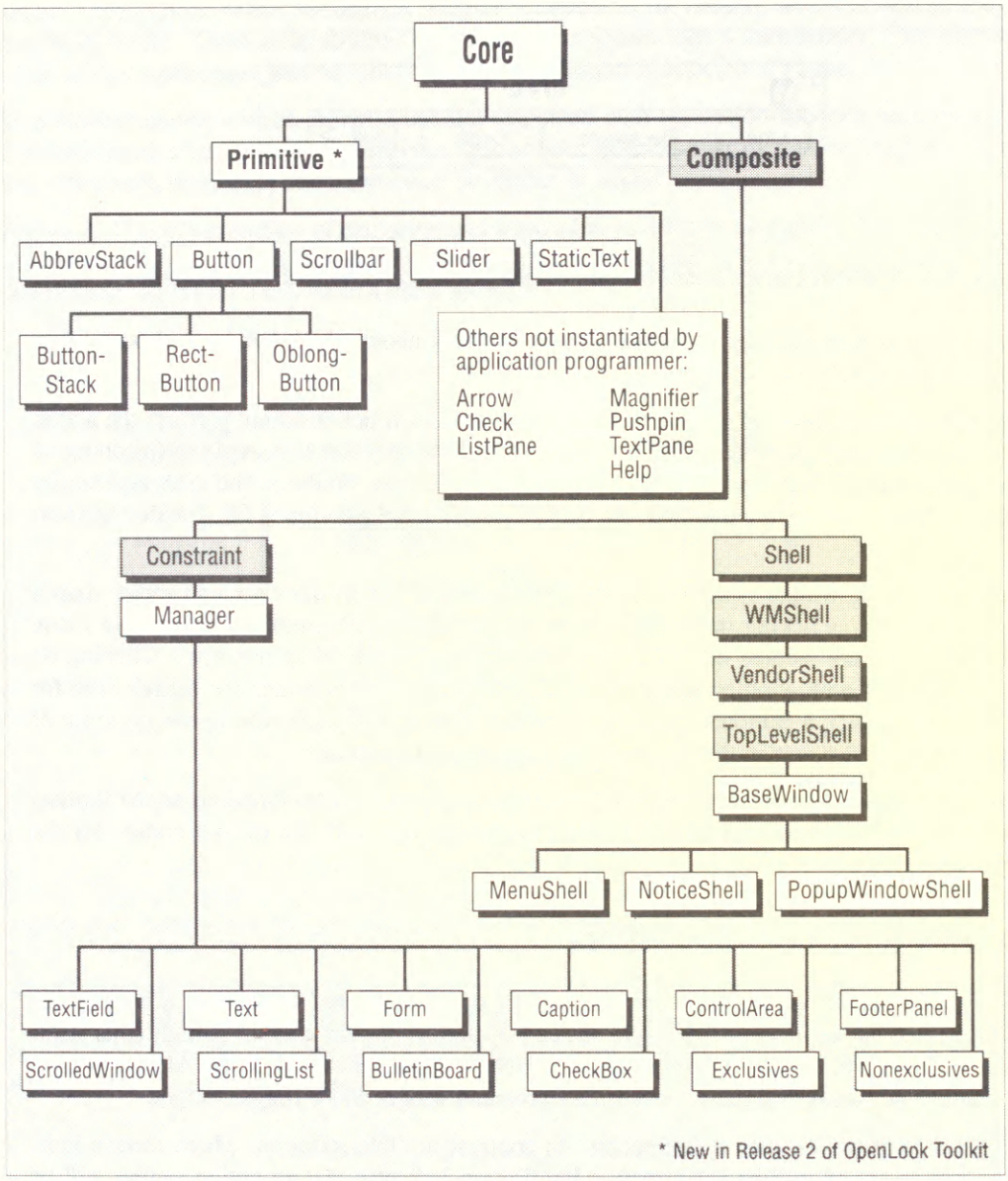


Figure A-2. Class inheritance hierarchy of the AT&T OPEN LOOK widgets

The OPEN LOOK widget set implements similar functions using the OblongButton and ButtonStack widgets. Figure A-3 shows a Control Area containing OblongButton and ButtonStack widgets.

The OblongButton widget provides many niceties lacking from the Athena Command widget. One of the most important is that OblongButton has a resource that allows one button among several to be designated as the default, in which case the button is bordered by a

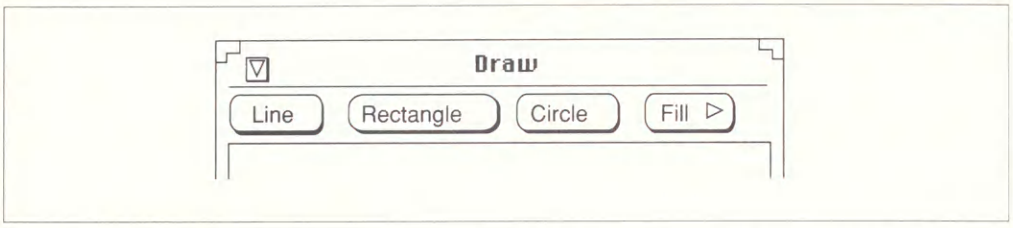


Figure A-3. An OPEN LOOK ControlArea with OblongButton and ButtonStack widgets

double line, to give the user immediate visual feedback about which button to choose when several are available.

The OblongButton also has the notion of a "busy" state in which it cannot perform the action because it is already doing it. (This is different from a widget that is insensitive (meaning its function is unavailable), though in practice the effect is quite similar.) The label and border of an insensitive button are dimmed; the background of a busy button is filled with a stippled pattern.

The ButtonStack widget is similar to the OblongButton but invokes a menu rather than a single command. Clicking on it with one mouse button (usually Button 3) pops up a menu (which may in turn include other ButtonStack widgets, for cascading pop ups.) Clicking on the ButtonStack with another mouse button (usually Button 1) activates the default item for the menu. Visually, a ButtonStack is differentiated from an OblongButton by the presence of an arrowhead that points in the direction that the menu will pop up.*

In addition, there is an AbbrevStack widget, which performs similar functions as the ButtonStack widget, but shows up only as a small unlabelled box with the default choice for the menu displayed beside it.

A.1.1.2 Exclusive and Nonexclusive Settings

The Athena Toggle widget provides the concept of a button that establishes a setting (for example, sets an application resource) instead of performing an action. It highlights itself when selected, but remains highlighted until selected again. However, in the Athena Widget set, there is no visual distinction between a Command widget and a Toggle widget.

OPEN LOOK provides such a distinction. In contrast to OblongButton, which always indicates that an action will be performed, a RectButton indicates that an option setting will be chosen. This setting may be exclusive or nonexclusive, as determined by the RectButton widget itself, depending on whether the widget is managed by an Exclusives or Nonexclusives composite widget. The RectButton widget class will not work correctly unless managed by one of these two composite widgets. The Exclusives and Nonexclusives widgets are themselves usually children of a Menu or ControlArea widget.

*In an earlier implementation of the toolkit, a ButtonStack widget had a double border on its bottom half, giving it the appearance of a stack of regular buttons. Hence its name, which is now merely historical.

In an Exclusives widget, RectButton widgets are laid out side by side in one or more columns. One or none of the RectButton widgets is chosen as the default, which is indicated by a double border. Once a RectButton is selected, it is shown with a dark border. The Exclusives widget makes sure that no more than one RectButton is selected at a time.

In a Nonexclusives widget, RectButtons are displayed with separation between each button. As when used in an Exclusives widget, a dark border indicates that the option has been chosen. However, more than one button may be chosen at a time.

Figure A-4 shows examples of exclusive and nonexclusive settings on menus. Note that, like the OblongButton, a RectButton may display a pixmap instead of a label. This makes the RectButton useful for a palette in a paint program.

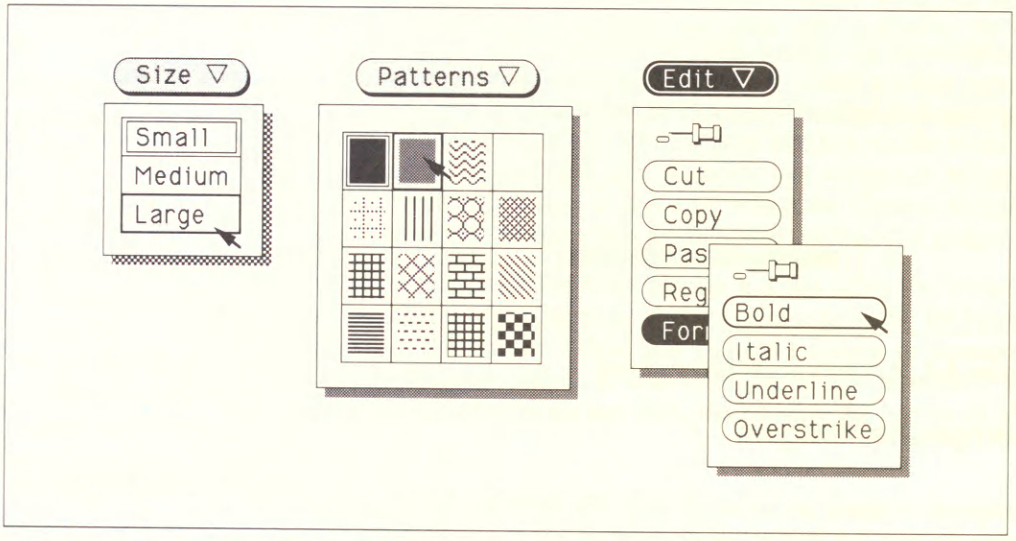


Figure A-4. OPEN LOOK RectButtons controlled by Exclusives and Nonexclusives widgets

The CheckBox widget provides an alternate way to display nonexclusive settings to the user. It displays a small box next to the label and displays a checkmark in the box when the option is selected. Checkboxes appear in ControlAreas rather than on menus. Figure A-5 shows examples of CheckBox widgets.

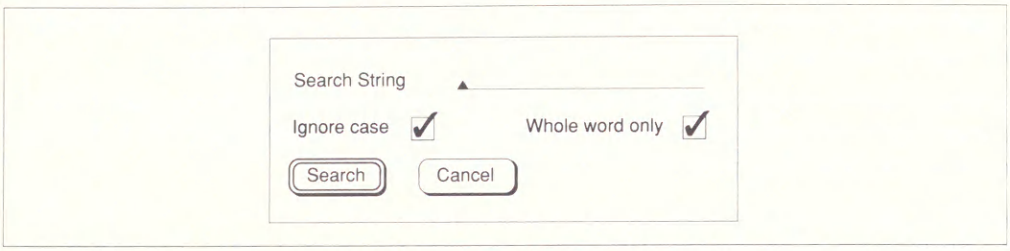


Figure A-5. An OPEN LOOK CheckBox widget

A.1.1.3 Analog Controls

In addition to the various kinds of buttons outlined above, OPEN LOOK provides an analog control called a Slider. A Slider widget is used analogously to a Scrollbar but is used for setting a numeric value. Figure A-6 shows a Slider widget.

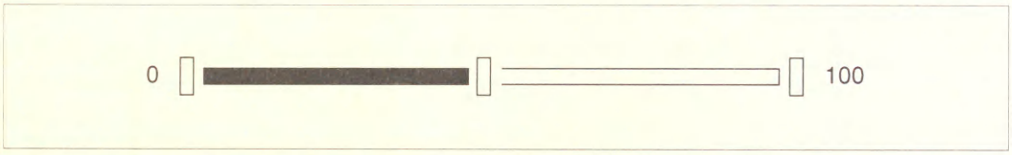


Figure A-6. An OPEN LOOK Slider widget

A.1.2 Composite Widgets

Composite widgets are in many ways the most important widgets in any widget set. They define the way that widgets work together, and they give consistency to an application.

A.1.2.1 Menus and Control Areas

As we've already discussed, command buttons of any kind are usually displayed as part of a menu or control area.

Menus can either pop up below a ButtonStack or an AbbrevStack, or if the button is itself displayed on a menu, to the right, in a menu cascade. Figure A-4 showed examples of menus.

The Menu widget is a pop-up widget created with `XtCreatePopupShell`. It has a single child, which is a ControlArea widget.

The ControlArea widget places its children in rows or columns. Resources allow the application to specify a fixed width and/or height, or a fixed number of rows or columns. ControlArea widgets are usually used as the parent of OblongButton, ButtonStack, Exclusives or Nonexclusives widgets (which in turn manage RectButton widgets, as described in the next section).

A.1.2.2 General Purpose Composite Widgets

We've already discussed the Composite widgets relating to control areas and menus. However, there are several general-purpose composite widgets in the OPEN LOOK set as well.

The BulletinBoard widget provides a free-form area for placing subwindows. Widgets can be placed on a BulletinBoard at arbitrary x and y coordinates; if no coordinates are specified, they appear in the upper left corner. The BulletinBoard provides no management of its children, and is often used to estabish the base frame for an application, since it allows the application programmer to place the major components of the application, rather than having to go by some Composite widget's arbitrary placement decisions.

A BulletinBoard is often used as the main window of an application.

The Form widget is a constraint widget similar to the Athena Form widget. It allows the placement of widgets to be specified relative to each other, and with rules governing their separation or relative position.

The Caption widget is like an Athena Label widget turned inside out. Like the Label widget, it prints a string. However, while the label widget's string is printed inside a visible widget border, a Caption string appears outside a bordered area. Caption is a composite widget class, and its label typically refers to a child widget of any size, which the Caption widget manages. The label can be aligned on either the right, left, top or bottom of the child widget.

The FooterPanel widget provides a consistent method for placing a footer along the bottom of another window. The footer panel takes two children. The top child is typically the main composite widget of the application; the bottom widget may contain a control or message area. The basic feature of the FooterPanel widget is that when the widget is resized, it applies all the change in the vertical direction to the top child, maintaining the bottom child at a constant height.

A.1.2.3 Scrollbars and Scrollable Windows

OPEN LOOK scrollbars use the visual metaphor of an elevator on a cable, but functionally they are similar to the Athena Scrollbar widget. The drag area (the thumb in an Athena Scrollbar widget) doesn't change size; instead, as shown in Figure A-7, there is a separate area that indicates the proportion of the data that is currently being displayed.

Scrollbars may be oriented either horizontally or vertically.

Scrollbars are used as a component of a ScrolledWindow widget, which, like the Athena Viewport widget, provides a scrollable view of a data area in a child widget. The child widget is typically larger than the view area, but only the area in the parent's view area can be seen at any one time. Figure A-1 showed a ScrolledWindow widget as the main application pane. The ScrollingList widget displays a scrollable list of editable text fields, and provides facilities for choosing and displaying one of the fields as "currently selected." Items can be selected from the list, changed, copied, and so on. This widget is useful for providing an interface to select a file for reading or writing.

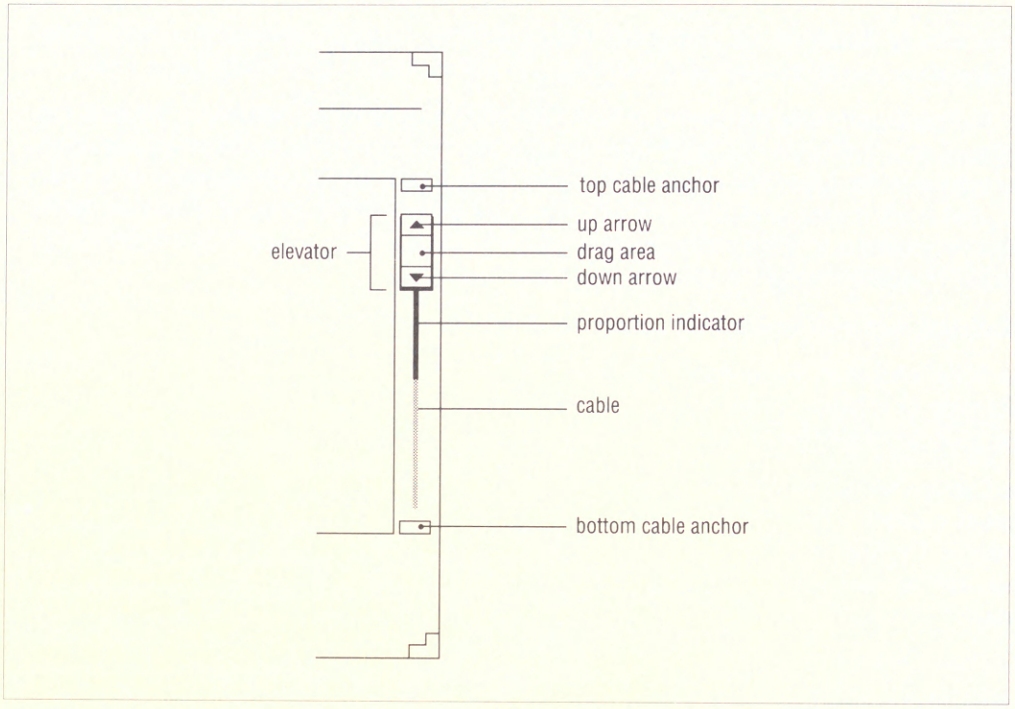


Figure A-7. An OPEN LOOK Scrollbar

Figure A-8 shows a ScrollingList widget.

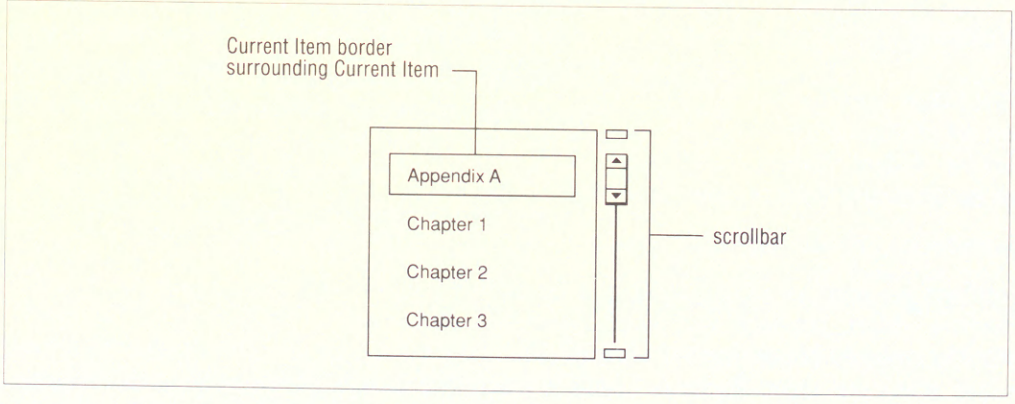


Figure A-8. An OPEN LOOK ScrollingList widget

A.1.3 Pop Ups

In addition to Menu widgets, the OPEN LOOK widget set contains three other special types of pop-up widgets: Notices, PopupWindows, and Help windows. A Notice is used to request confirmation or other information from the user. The widget contains a text area, where the message to the user is displayed, and a control area containing one or more buttons, one of which must be the default button.

Figure A-9 shows an OPEN LOOK Notice widget.

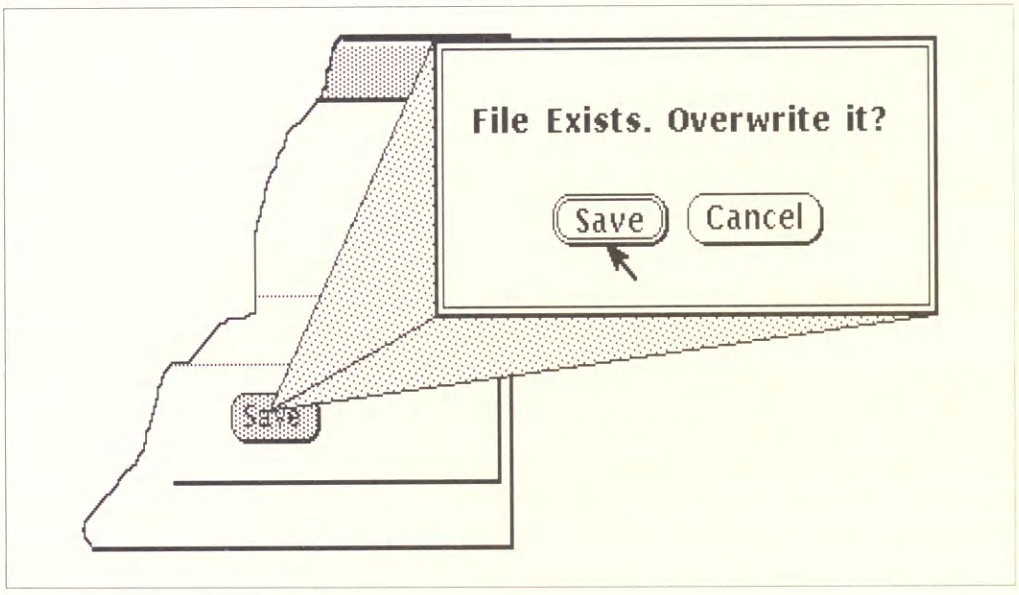


Figure A-9. An OPEN LOOK Notice

A Notice grabs the pointer. The only input allowed is to the Notice. Once the user has clicked a button, the Notice disappears.

The second special pop-up type is a PopupWindow, which can be used for more complex pop ups. Unlike a Notice, which is a subclass of OverrideShell, a PopupWindowShell is a subclass of WMShell, and so is decorated by the window manager. It has all the visual attributes of a top-level window, including resize corners, etc. In addition, it displays a pushpin in the upper left corner. If the user clicks on the pushpin with the pointer, the menu doesn't go away when its action has been performed, but stays on the screen. This allows the user to keep menus (and other frequently-referenced pop ups, such as help screens) "pinned" on the display, where they can be moved like regular windows. (Menus can also display a pushpin; its presence or absence is controlled by a widget resource.)

A PopupWindow typically contains an upper control area that may include menus, and a lower control area that may be used for buttons invoking widget actions. Resources allow for automatic creation of several buttons, including a "reset to factory" button, a "set defaults" button, and several other ways of setting standard properties for an application.

Figure A-10 shows a PopupWindow.

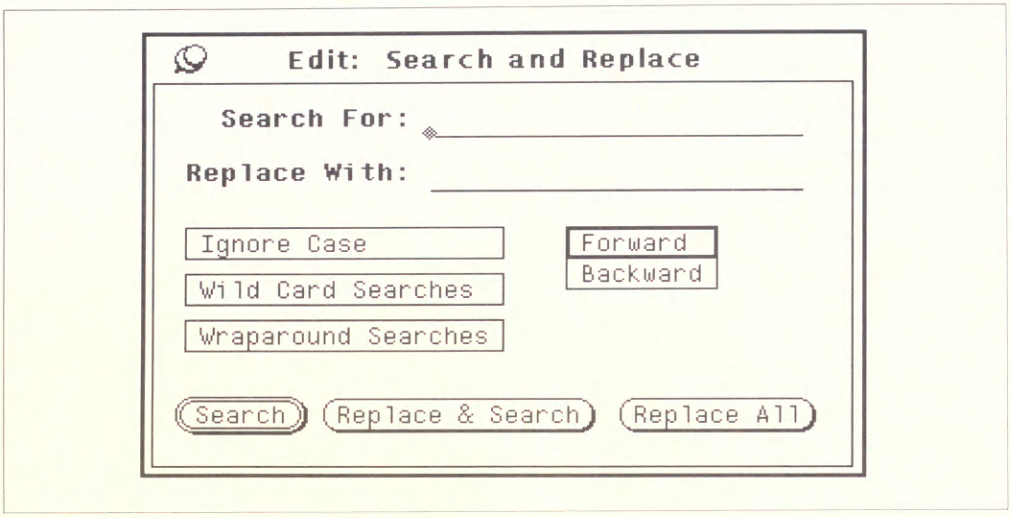


Figure A-10. An OPEN LOOK PopupWindow

A Help window is not instantiated in the usual way. Instead, an application uses the `OlRegisterHelp` function to register help text with the toolkit. Text can be associated with a widget class, a widget instance, or a window. When the user clicks the second pointer button on an object, the Help widget is automatically displayed by the toolkit. The Help widget includes a Magnifier subwidget, which displays a magnifying glass containing an image of the part of the screen on which the user clicked the pointer. Figure A-11 shows a Help window.

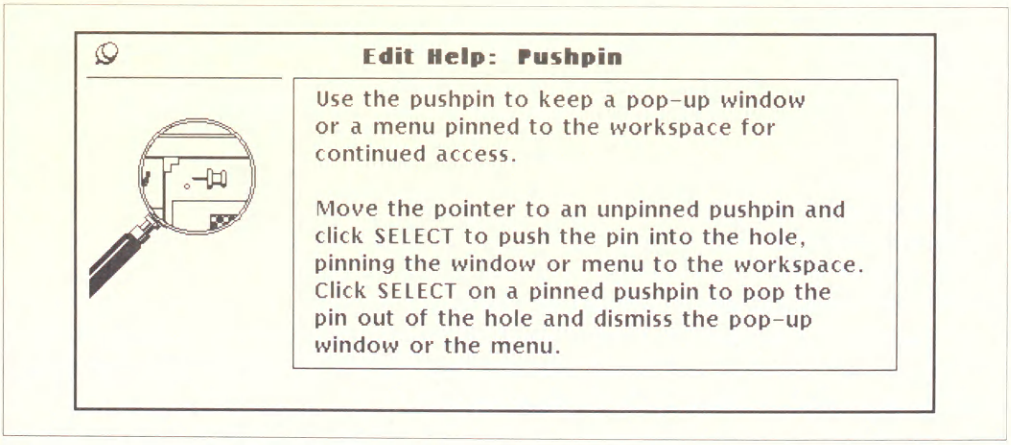


Figure A-11. An OPEN LOOK Help window

A.1.4 Text Widgets

OPEN LOOK offers three separate text widgets. The StaticText widget is similar to the Athena Label widget in that it displays a fixed text string. However, it can handle longer strings by using wordwrapping to break the string onto multiple lines.

The OPEN LOOK Text widget is very similar to the Athena Text widget. It provides a general-purpose editable text widget, with editing commands customizable via a translation table.

One of the annoying weaknesses of the Athena Text widget is that it is difficult and inefficient to use as a single-line editable field. (A program can add translations for the Return key to limit the text to a single line.) OPEN LOOK addresses this need with the TextField widget, useful for developing form-driven applications. TextField widgets were shown in Figure A-11.

The TextField widget provides simple editing commands, and scrolling if the string is too long to be displayed. When the keyboard focus leaves the widget, or when the Tab or Return key is pressed, it passes the data in the field to the application for validation.

A.1.5 Drawing Areas

Like the Athena widget set, the AT&T OPEN LOOK widgets provide no widget explicitly labeled as a drawing area. As described in this book, one is expected either to create a custom widget for an application's main window, or to use a very basic widget class, and add actions for drawing.

The AT&T OPEN LOOK widget set does include a Stub widget class (which is not documented in the manual, but is included in the source), which is useful for providing a window for drawing.

A.2 The OSF/Motif Widgets

Figure A-12 shows the general look of a Motif application, the Motif Reference Editor, *mre*. *mre*, developed by Mitch Trachtenberg, is OSF demo software, which is available as part of the Motif distribution.

As with the AT&T OPEN LOOK widget set, some of the features of a main application window are actually decoration provided by the window manager, *mwm*. As shown in Figure A-11, these include the title bar, which displays:

- The title provided by the application.

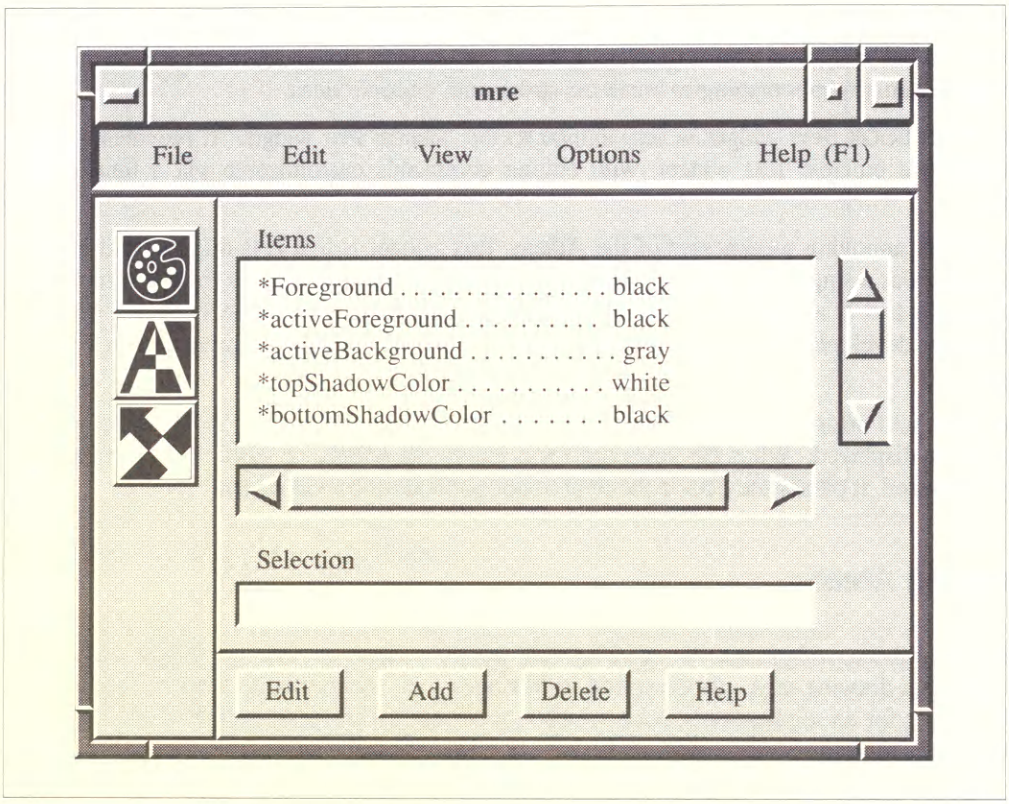


Figure A-12. Look and feel of a Motif application

- A Menu button, which drops down a menu containing basic window manipulation commands.
- A "Minimize" button, which iconifies the application.
- A "Maximize" button, which causes the application window to fill the entire screen.

You will also recognize Motif's version of many of the common controls discussed in the section on the AT&T OPEN LOOK widget set.

Figure A-13 shows the inheritance hierarchy of the Motif widgets. The Intrinsics-supplied widget classes are shaded grey.

The Primitive and Manager widgets are not generally instantiated and exist only to provide resources and other features inherited by other widgets.

In addition, Motif supports a Gadget class, which, as described in Chapter 12, is subclassed from RectObj rather than Core. Gadget equivalents exist for the Label widget, some classes of Button widgets, and the Separator widget. Gadgets are not shown in Figure A-13.

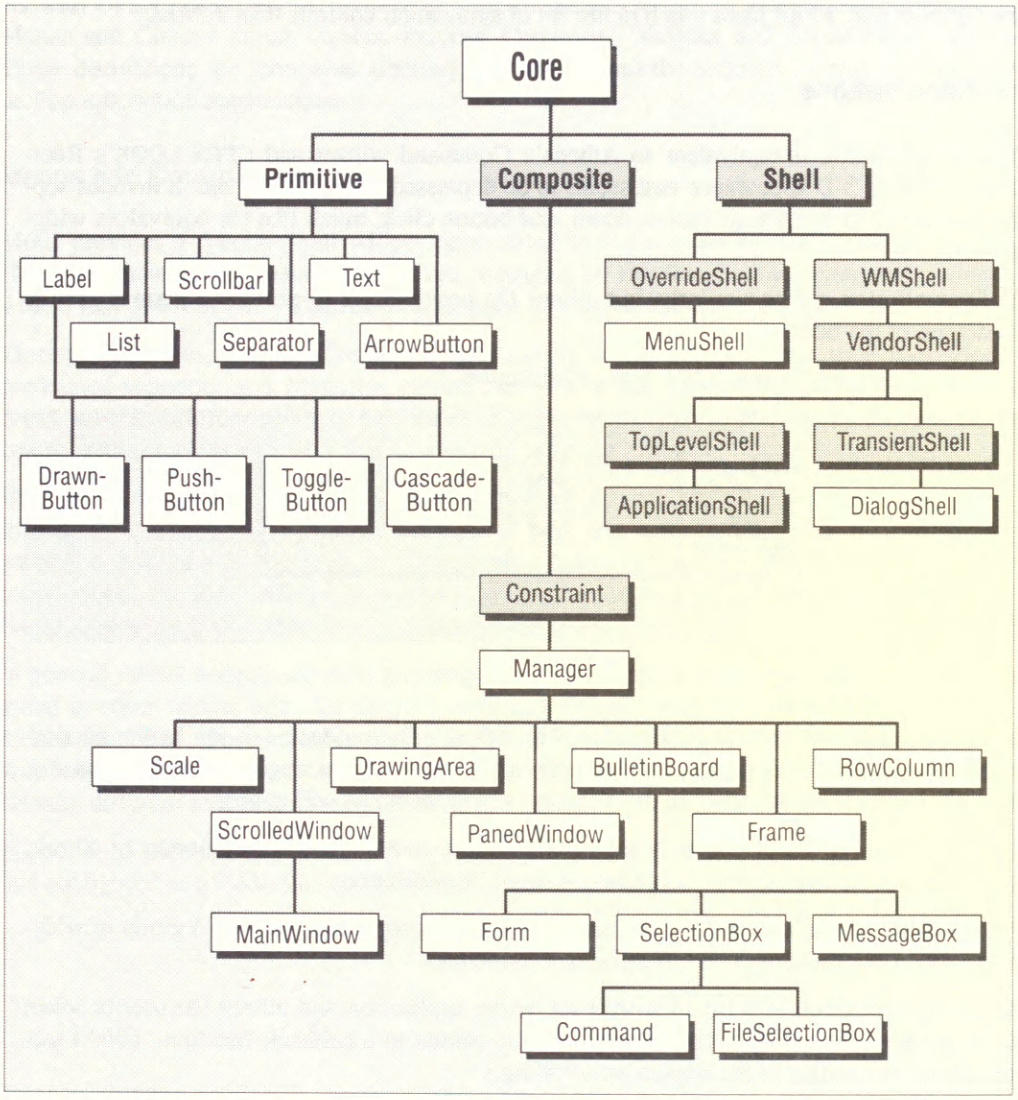


Figure A-13. Class inheritance hierarchy of the Motif widget set

A.2.1 Application Controls

Like OPEN LOOK, Motif has a much richer set of application controls than Athena.

A.2.1.1 Command Buttons

Motif's PushButton is equivalent to Athena's Command widget and OPEN LOOK's RectButton. It has a 3-D appearance and seems to be depressed when clicked on. It invokes separate callbacks for button up, button down, and button click, much like the equivalent widgets in other sets.

The DrawnButton works similarly, but allows the programmer to provide a pixmap for the appearance of the button.

Figure A-14 shows a DrawnButton and a PushButton.

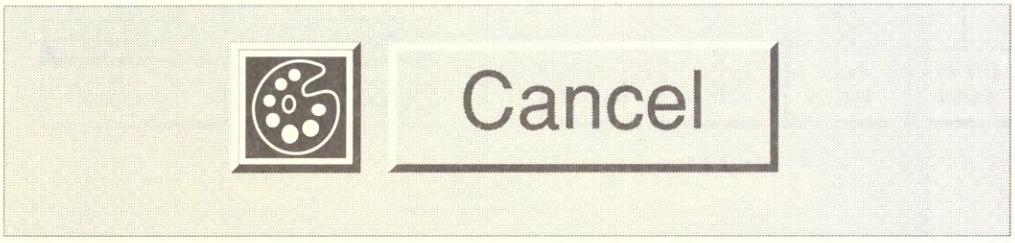


Figure A-14. Motif DrawnButton and PushButton widgets

The CascadeButton is similar in effect to OPEN LOOK's ButtonStack—it can have a particular appearance that indicates that a menu is invoked, rather than a single callback. Typically, this is simply an arrow pointing in the direction where the menu will appear.

A ToggleButton is used for option setting, much like OPEN LOOK's RectButton or CheckBox. Figure A-16 shows a box containing a set of ToggleButtons.

The Separator widget can be used to draw a line or other separator between a group of widgets in a menu or other box. It is typically used in menus.

The List widget displays a list of strings set by the application and allows the user to select one or more of the strings. The selected data is passed to a callback function. (We'll talk more about this widget in the section on scrolling.)

A.2.1.2 Analog Controls

Motif's Scale widget is similar to OPEN LOOK's Slider but is more powerful since it can be used to display as well as to control analog values.

A.2.2 Composite Widgets

As with OPEN LOOK, we've divided the discussion of Composite widgets into three areas: Menus and Control Areas, General-Purpose Composite Widgets, and Scrollable Windows. These distinctions are somewhat arbitrary and with menus the sections overlap with the one on Pop ups, which appears later.

A.2.2.1 Menus and Control Areas

Motif provides a special Shell widget class called MenuShell for managing pop-up menus. However, most actual menu displays are managed by the RowColumn composite widget, which, like OPEN LOOK's ControlArea, displays buttons in rows or columns.

Through resources, the RowColumn widget can be configured to create such specialized, predefined elements as a MenuBar (which can only accept CascadeButton widgets as children), several different styles of pull-down or pop-up menu panes, and several preconfigured control areas, such as a "Radio Box" containing multiple exclusive ToggleButton gadgets.

Here you can begin to see the wide divergence in programming style made possible by the Xt Intrinsics. It is possible to create a hierarchy of relatively simple widgets to perform separate parts of a task, or a single, very complex widget which is highly configurable. In one of its incarnations, the RowColumn widget is equivalent to an OPEN LOOK ControlArea plus an Exclusives widget; in another, a ControlArea plus a Nonexclusives.

In general, Motif widgets are more complex and have many more resources than widgets provided in other widget sets. To simplify their use, though, Motif provides numerous convenience functions. For example, `XmCreateRadioBox` will create a RowColumn widget with one specialized set of resources, while `XmCreateMenuBar` will create one that is entirely different in appearance and function.

Figure A-15 shows a RowColumn widget configured as a MenuBar and Figure A-16 shows one configured as a RadioBox (each with appropriate children).

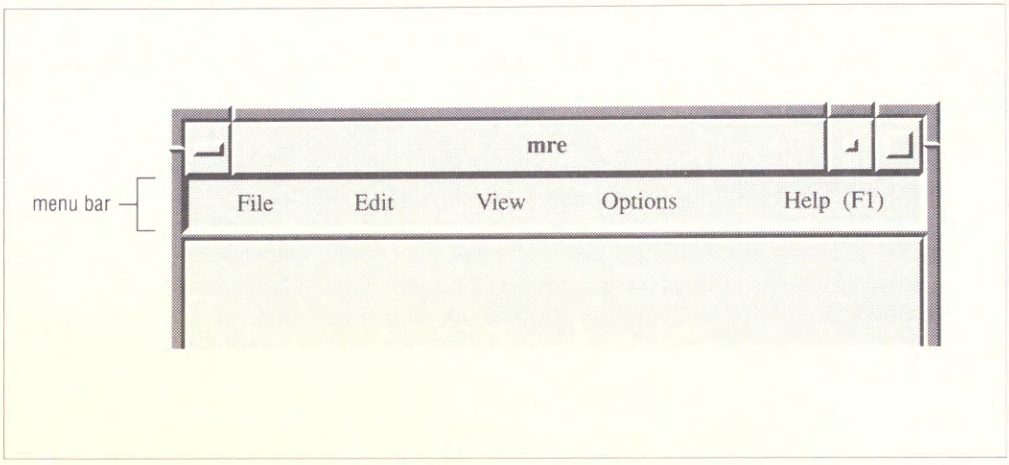


Figure A-15. A Motif RowColumn widget configured as a Menu Bar

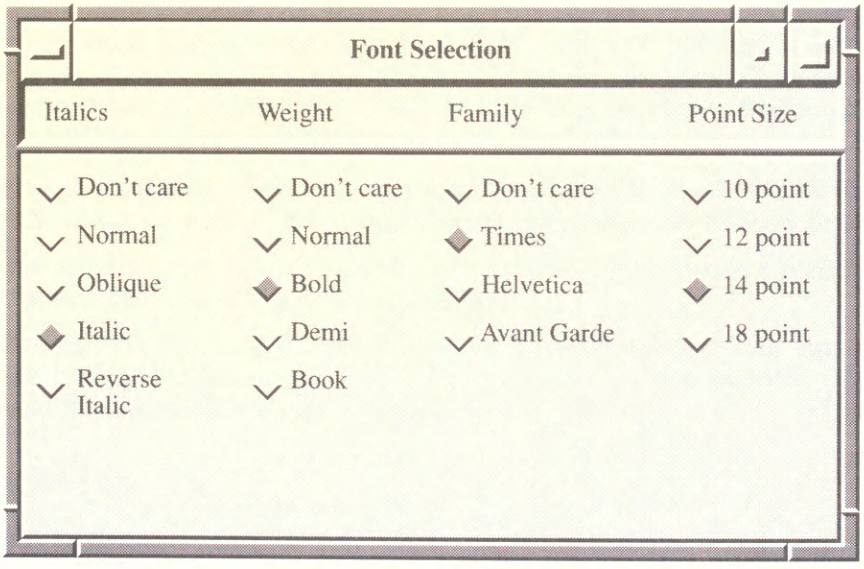


Figure A-16. A Motif RowColumn widget configured with four Radio Boxes

Figure A-17 shows a RowColumn widget implementing a drop-down menu.

Items on Motif menus can be selected by dragging the pointer down the menu and releasing it when the chosen item is highlighted. Alternately, the pointer can simply be clicked on the menu title to drop down the menu. Clicking on an item in the menu selects it; clicking anywhere other than in the menu pops the menu down without executing any item.

Note also that as a general feature, Motif menus support accelerators. That is, there are keyboard equivalents for every menu item. These keyboard accelerators are listed after the menu label, as shown above. In addition, typing the underlined letter in any menu item label when the pointer is in the menu will select that menu item. These underlined letters are called "mnemonics."

The items on a menu bar or menu pane simply appear as labels but when selected take on the 3-D appearance of a PushButton.

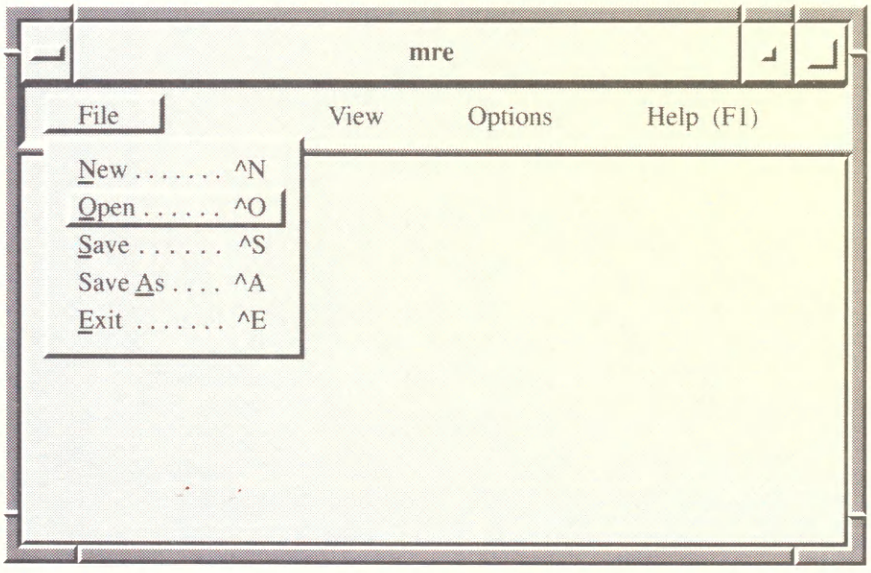


Figure A-17. A Motif RowColumn widget configured as a drop-down menu

A.2.2.2 General Purpose Composite Widgets

The BulletinBoard widget provides simple composite management, allowing widgets to be placed arbitrarily anywhere within its confines. The only constraint is that they are not allowed to overlap.

The Form widget is a subclass of BulletinBoard that, like the widgets of the same name in other sets, allows children to be laid out relative to each other or to one or another of the sides of the Form. The children will thus always maintain their proper relative position when the application is resized.

Figure A-18 shows a fully configured Form.

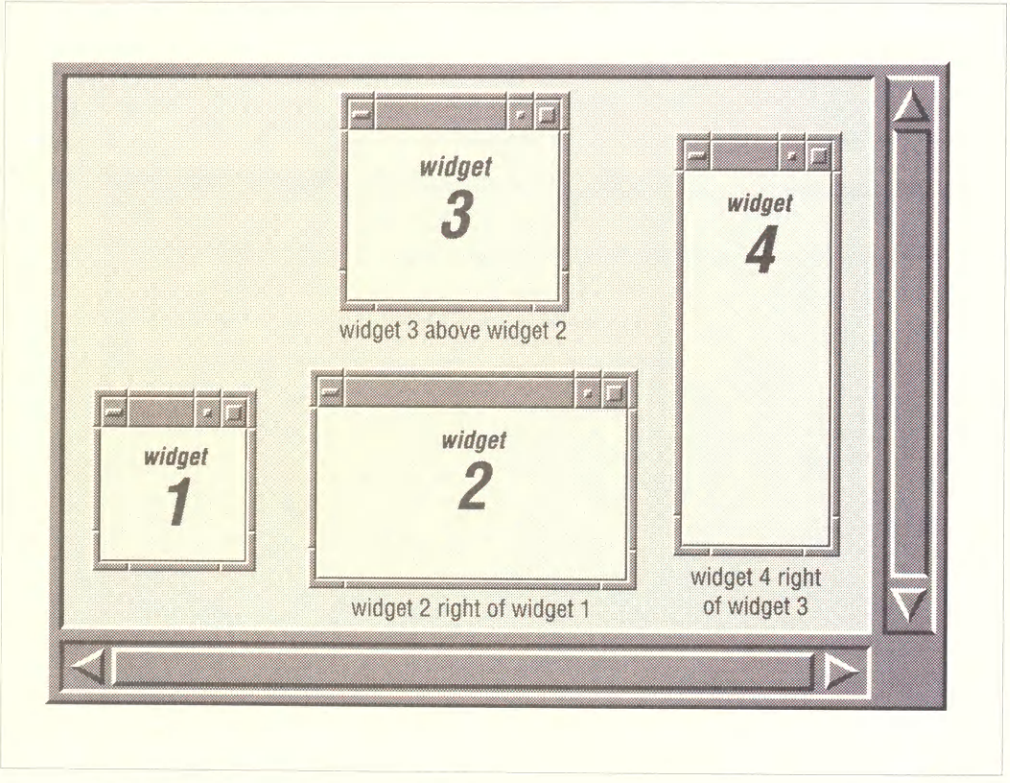


Figure A-18. A Motif Form widget and children

The Frame widget is used simply to provide a consistent border for widgets that might not otherwise have one. One use is to give a RowColumn widget a border with a 3-D appearance.

A.2.2.3 Scrollable Windows

A Motif ScrollBar is illustrated in Figure A-19.

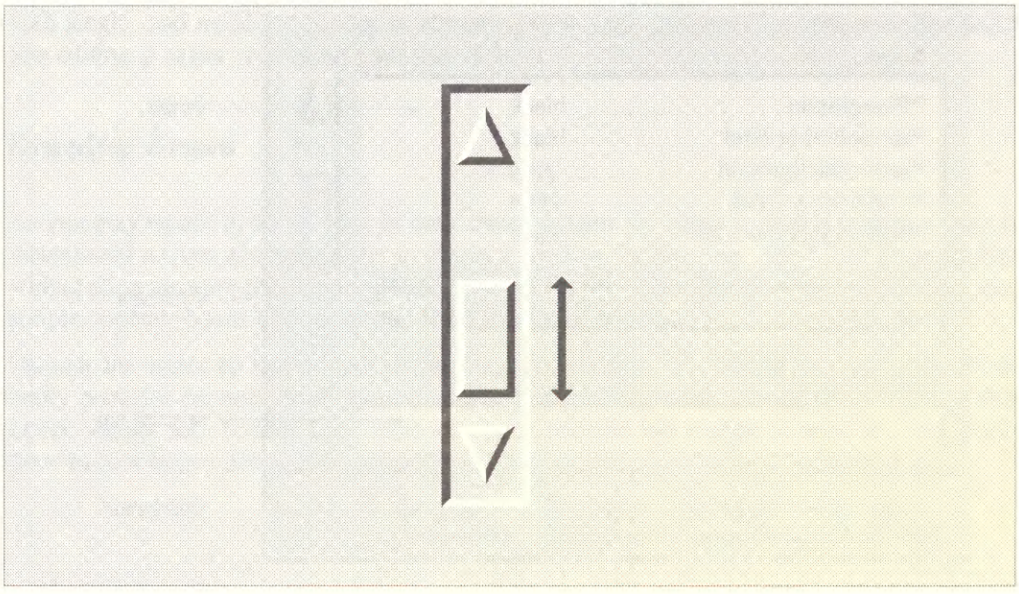


Figure A-19. A Motif ScrollBar

Like the Athena Scrollbar widget, the scrollbar has a "thumb" or slider that can be dragged up and down to scroll the associated window. You can also click above or below the thumb to move it a screenful at a time. Unlike the Athena widget, it also displays arrows at either end that can be used to scroll line by line. The associated window scrolls in the indicated direction as long as the pointer button is held down in one of the arrows.

There are several different types of scrolling windows. The ScrolledWindow widget, like Athena's Viewport, provides a general mechanism for attaching scrollbars to some other window.

The MainWindow widget is a subclass of ScrolledWindow with a special appearance reserved for application main windows. Figure A-14 showed a MainWindow widget.

Using the `XmCreateScrolledList` function, a List widget can be created as a child of a ScrolledWindow, giving the effect of a simple scrolled list. In addition, there are several flavors of more complex scrolling lists. These include the SelectionBox widget, and its two subclasses, Command and FileSelectionBox.

A general-purpose SelectionBox is akin to a ScrolledWindow/List combination, but adds a Text widget for entering additional data not on the list. The SelectionBox also adds at least three buttons labeled by default *OK*, *Cancel* and *Help*.

Figure A-20 shows a SelectionBox.

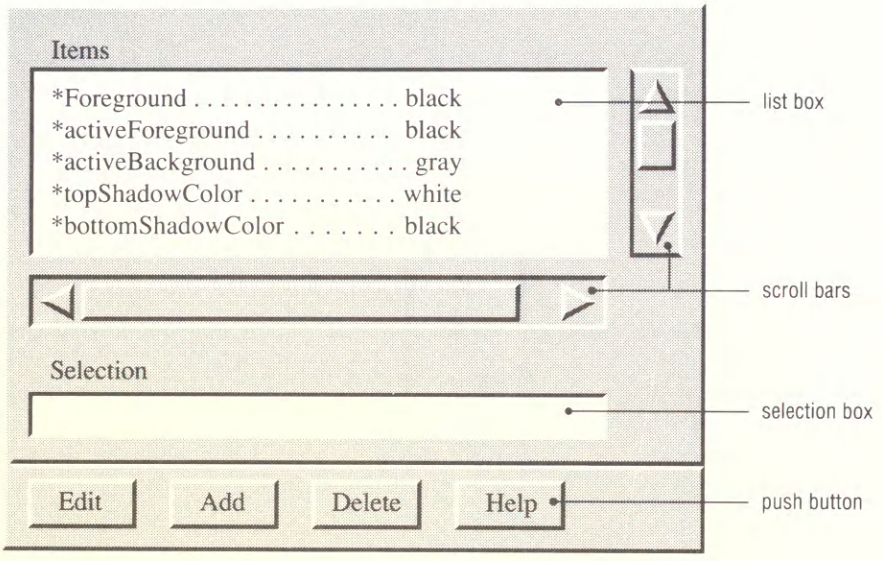


Figure A-20. A Motif SelectionBox

A FileSelectionBox is a SelectionBox specially designed to present a list of filenames for selections. A Command widget is a special kind of SelectionBox whose list consists of the history of commands entered in the Text widget. Each time a new command is entered, it is added to the history list.

A.2.3 Pop Ups

Motif defines two classes of Shell widgets: DialogShell, which is used for parenting Dialog boxes, and MenuShell, which is used for menus. These classes are rarely instantiated directly, but are instead created by convenience functions that also create their composite children.

For example, functions exist to create a DialogShell with a variety of pre-configured MessageBox widgets as the visible child.

As we've already discussed, a specially configured RowColumn widget is used to create a menu pane as the visible child of a MenuShell widget.

A.2.4 Text Widgets

Like Athena and the AT&T OPEN LOOK widgets, Motif provides a Text widget that supports a complete editing command set. Like Athena, and unlike AT&T's OPEN LOOK widget set, both single- and multiline editing is supported by a single widget. But arranging for single-line editing is easier with Motif than with Athena.

A.2.5 Drawing Areas

As you may recall, to do drawing in the Athena widgets we either created a custom widget or instantiated a Core widget in order to obtain a window for drawing. The Motif DrawingArea widget class answers this need in Motif. It provides a window for drawing and provides very simple, bulletin-board like composite management of children.

Though the name of this widget class sounds promising, you should be aware that Motif really provides no more sophisticated drawing capabilities than Athena or the AT&T OPEN LOOK widget set. In each case, once you have selected the widget to draw on, you simply draw in its window using Xlib calls.

B

Specifying Fonts and Colors

This appendix describes the possible values for color, font, and geometry resource specifications.

In This Chapter:

Color Specification 447
- Color Names 447
- Hexadecimal Color Specification 448
  - The RGB Color Model 449
  - How Many Colors are Available? 450

Font Specification 452
- Font Naming Conventions 453
- Font Name Wildcarding 454
- Font Name Aliasing 456
- Making the Server Aware of Aliases 457
- The fonts.dir Files 457

Window Geometry 458

B
Specifying Fonts and Colors

This appendix describes the possible values for color, font, and geometry resource specifications.

B.1 Color Specification

Many clients have resources and command-line options that allow you to specify the color of the window background, foreground (the color that text or graphic elements will be drawn in), or window border. For example, the following resources might be set for a Label widget:

```
*background:    orange          set the background color to orange
*foreground:    black           set the foreground color to black
*borderColor:   black           This must be Halloween!
```

The corresponding command-line options have the form:

-bg `color` sets the background color

-fg `color` sets the foreground color

-bd `color` sets the border color

Some clients allow additional options to specify color for other elements, such as the cursor, highlighting, and so on.

By default, the background is usually white and the foreground black, even on color workstations. You can specify a new color using either the names in the X Window System's color name database or hexadecimal values.

B.1.1 Color Names

The *rgb.txt* file, usually located in */usr/lib/X11* on UNIX systems, is supplied with X and consists of predefined colors assigned to specific (but not necessarily intuitive) names.*

*A corresponding compiled file called *rgb.pag* contains the definitions used by the server; the *rgb.txt* file is the human-readable equivalent.

The following are the default color names that come with the X Window System. (See Appendix A, *System Management*, in Volume Three, *X Window System User's Guide*, for information on customizing color name definitions.) This file is not part of the X standard, so vendors are free to modify it. However, most will just add to it, or redefine the values associated with each color name for better effects on their display hardware.

```
aquamarine          mediumaquamarine    black               blue
cadetblue           cornflowerblue      darkslateblue       lightblue
lightsteelblue      mediumblue          mediumslateblue     midnightblue
navyblue            navy                skyblue             slateblue
steelblue           coral               cyan                firebrick
gold                goldenrod           mediumgoldenrod     green
darkgreen           darkolivegreen      forestgreen         limegreen
mediumforestgreen   mediumseagreen      mediumspringgreen   palegreen
seagreen            springgreen         yellowgreen         darkslategrey
darkslategray       dimgrey             dimgray             lightgrey
lightgray           khaki               magenta             maroon
orange              orchid              darkorchid          mediumorchid
pink                plum                red                 indianred
mediumvioletred     orangered           violetred           salmon
sienna              tan                 thistle             turquoise
darkturquoise       mediumturquoise     violet              blueviolet
wheat               white               yellow              greenyellow
```

As of R4, a number zero through three can be appended to each of these names in order to get various intensities of each color. In addition, a complete range of grays are provided by using the name *gray* or *grey* followed by a number from zero through 100. There are also a few colors not provided in the R3 database, such as *snow* and *misty rose*. These *names can be used directly when the specific color is wanted.*

For example, the command line:

```
% xterm -bg lightblue -fg darkslategray -bd plum &
```

creates an *xterm* window with a background of light blue, foreground of dark slate gray, and border of plum. Note that the RGB values in the color database provided by MIT are correct for only one type of display; you may find that the color you get is not exactly what you expect given the name. To combat this, vendors may have corrected the RGB values to give colors closer to what the name implies.

At the command line, a color name should be typed as a single word (for example, `darkslategray`). However, you can type the words comprising a color name separately if you enclose them in quotes, as in the following command line:

```
% xterm -bg "light blue" -fg "dark slate gray" -bd plum &
```

B.1.2 Hexadecimal Color Specification

You can also specify colors more exactly using a hexadecimal color string. You probably won't use this method unless you require a color not available by using a color name. Moreover, you shouldn't use this method unless necessary because it tends to discourage the shar-

ing of colors between applications. In order to understand how this works, you may need a little background on how color is implemented on most workstations.

B.1.2.1 The RGB Color Model

Most color displays on the market today are based on the RGB color model. Each pixel on the screen is actually made up of three phosphors: one red, one green, and one blue. Each of these three phosphors is excited by a separate electron beam. When all three phosphors are fully illuminated, the pixel appears white to the human eye. When all three are dark, the pixel appears black. When the illumination of each primary color varies, the three phosphors generate a subtractive color. For example, equal portions of red and green, with no admixture of blue, makes yellow.

As you might guess, the intensity of each primary color is controlled by a three-part digital value—and it is the exact makeup of this value that the hexadecimal specification allows you to set.

Depending on the underlying hardware, different servers may use a larger or smaller number of bits (from 4 to 16 bits) to describe the intensity of each primary. To insulate you from this variation, clients are designed to take color values containing anywhere from 4 to 16 bits (1 to 4 hex digits), and the server then scales them to the hardware. As a result, you can specify hexadecimal values in any one of the following formats:

```
#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB
```

where R, G, and B represent single hexadecimal digits and determine the intensity of the red, green, and blue primaries that make up each color.

When fewer than four digits are used, they represent the most significant bits of the value. For example, `#3a6` is the same as `#3000a0006000`.

What this means concretely is perhaps best illustrated by looking at the values that correspond to some colors in the color name database. We'll use 8-bit values (two hexadecimal digits for each primary) because that is the way they are defined in the *rgb.txt* file:

```
#000000      black
#FCFCFC      white
#FF0000      red
#00FF00      green
#0000FF      blue
#FFFF00      yellow
#00FFFF      cyan
#FF00FF      magenta
#5F9F9F      cadet blue
#42426F      cornflower blue
#BFD8D8      light blue
#8F8FBC      light steel blue
#3232CC      medium blue
#23238E      navy blue
#3299CC      sky blue
```

```
#007FFF        slate blue
#236B8E        steel blue
```

As you can see from the colors given above, pure red, green, and blue result from the corresponding bits being turned full on. All primaries off yields black, while all nearly full on gives white. Yellow, cyan, and magenta can be created by pairing two of the other primaries at full intensity. The various shades of blue shown above are created by varying the intensity of each primary—sometimes in unexpected ways.

The bottom line here is that if you don't intimately know the physics of color, the best you can do is to look up existing colors from the color name database and experiment with them by varying one or more of the primaries till you find a color you like. Unless you need precise colors, you are probably better off using color names.

If you do specify a color using a hexadecimal value, try to use the same value for several applications so that they will share a color call.

In any event, using hexadecimal values for colors is not generally recommended, since it discourages sharing of color cells.

B.1.2.2 How Many Colors are Available?

The number of distinct colors available on the screen at any one time depends on the amount of memory available for color specification.

A color display uses multiple bits per pixel (also referred to as multiple planes or the *depth* of the display) to select colors. Programs that draw in color use the value of these bits as a pointer to a lookup table called a *colormap*, in which each entry (or *colorcell*) contains the RGB values for a particular color.* As shown in Figure B-1, any given pixel value is used as an index into this table—for example, a pixel value of 16 will select the sixteenth colorcell.

This implementation explains several issues that you might encounter in working with color displays.

First, the range of colors possible on the display is a function of the number of bits available in the colormap for RGB specification. If 8 bits are available for each primary, then the range of possible colors is 256^3 (somewhere over 16 million colors). This means that you can create incredibly precise differences between colors.

However, the number of different colors that can be displayed on the screen at any one time is a function of the number of planes. A four-plane system can index 2^4 colorcells (16 distinct colors); an eight-plane system can index 2^8 colorcells (256 distinct colors); and a 24-plane system can index 2^{24} colorcells (over 16 million distinct colors).

If you are using a four-plane workstation, the fact that you can precisely define hundreds of different shades of blue is far less significant than the fact that you can't use them all at the same time. There isn't space for all of them to be stored in the colormap at one time.

*There is a type of high-performance display in which pixel values are used directly to control the illumination of the red, green, and blue phosphors, but far more commonly, the bits per pixel are used indirectly, with the actual color values specified independently, as described here.

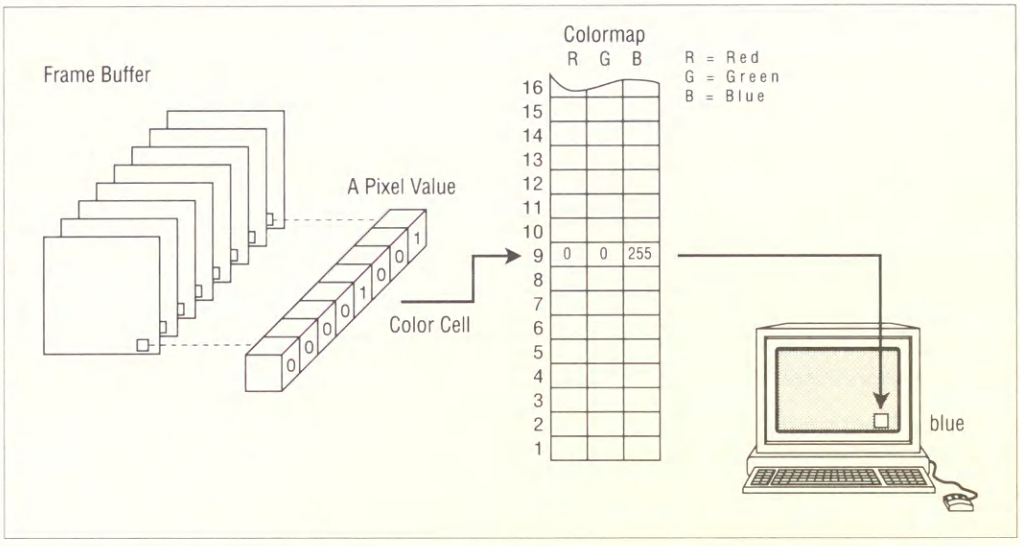


Figure B-1. Multiple planes used to index a colormap

This limitation is made more significant by the fact that X is a multiclient environment. When X starts up, usually no colors are loaded into the colormap. As clients are invoked, certain of these cells are allocated. But when all of the free colorcells are used up, it is no longer possible to request new colors. When this happens, you will usually be given the closest possible color from those that have already been allocated. However, you may instead be given an error message and told that there are no free colorcells.

In order to minimize the chance of running out of colorcells, many programs use "shared" colorcells. Shared colorcells can be used by any number of applications, but they can't be changed by any of them. They can be deallocated only by each application that uses them, and when all applications have deallocated the cell, it is available for setting again. Shared cells are most often used for background, border, and cursor colors.

Alternately, some clients have to be able to change the color of graphics they have already drawn. This requires another kind of cell, called private, which can't be shared. A typical use of a private cell would be for the palette of a color mixing application. Such a program might have three bars of each primary color, and a box which shows the mixed color. The primary bars would use shared cells, while the mixed color box would use a private cell.

In summary, some programs define colorcells to be read-only and shareable, while others define colorcells to be read/write and private.

To top it off, there are even clients that may temporarily swap in a whole private colormap of their own. Because of the way color is implemented, if this happens, all other applications will be displayed in unexpected colors.

In order to minimize such conflicts, you should request precise colors only when necessary. By preference, use color names or hexadecimal specifications that you specified for other applications.

For more information on color, see Chapter 7, *Color*, in Volume One, *Xlib Programming Manual*.

B.2 Font Specification

Most widgets that display text allow you to specify the font to be used in displaying text in the widget, via either the `XtNfont` resource, or the *-fn* and *-font* command-line options.

The X Window System supports many different display fonts, with different sizes and type styles. (These are *screen* fonts and are not to be confused with *printer* fonts.)

For Release 3 and subsequent releases, Adobe Systems, Inc., and Digital Equipment Corporation jointly contributed five families of screen fonts (Courier, Helvetica, New Century Schoolbook, Symbol and Times) in a variety of sizes, styles, and weights for 75 dots per inch monitors. Bitstream, Inc. contributed its Charter font family in the same sizes, styles, and weights for both 75 and 100 dots per inch monitors.

Most Release 2 fonts have been moved to the user-contributed distribution (*/usr/X11r3/contrib/fonts*).

In Release 3 and later, fonts are stored in three directories:

Directory	Contents
/usr/lib/X11/fonts/misc	Six fixed-width fonts (also available in Release 2), the cursor font.
/usr/lib/X11/fonts/75dpi	Fixed- and variable-width fonts, 75 dots per inch.
/usr/lib/X11/fonts/100dpi	Fixed- and variable-width fonts, 100 dots per inch.

These three directories (in this order) comprise X's default font path. The font path can be changed with the *fp* option to the *xset* client, as described in Volume Three, *X Window System User's Guide*. (The font path, together with a great deal of other information about the server defaults, can be listed with *xset query*.) All fonts in the font path can be listed with *xlsfonts*, and the characters in a font can be displayed on the screen with *xfd*.

The names of each font file in the font directories has a filename extension of *.snf*, which stands for *server natural format*. Fonts are distributed in *binary distribution format* (*bdf*), and may need to be adapted for a given server.

B.2.1 Font Naming Conventions

In Release 2, font names were determined by the names of the files in which they are stored, without the *.snf* extension. For example, the file *fg-16.snf* contains the font named fg-16.

If you do a listing of any of the current font directories, you'll notice that the filenames also have *.snf* extensions. However, font names are not determined by the names of the files in which they are stored.

Now, a font's name is determined by the contents of the font property named FONT* rather than the name of the file in which the font is stored.

If you run *xlsfonts*, you'll get an intimidating list of names similar to the one shown in Figure B-2, which upon closer examination contains a great deal of useful information:

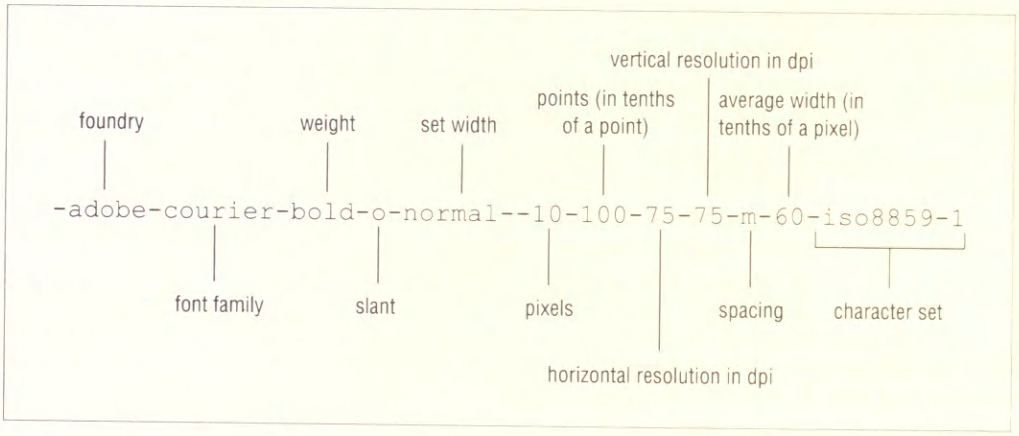


Figure B-2. A font name

This rather verbose line is actually the name of the font stored in the file *courBO10* (in the *75dpi* directory). This font name specifies the foundry (Adobe), the font family (Courier), weight (bold), slant (Oblique), set width (normal), size of the font in pixels (10), size of the font in tenths of a point (100—measured in tenths of a point, thus equals 10 points), horizontal resolution (75dpi), vertical resolution (75dpi), spacing (m, for monospace), average width in tenths of a pixel (60—measured in tenths of a pixel, thus equals 6 pixels) and character set (iso8859-1).

The meaning of many of these statistics is obvious. Some of the less obvious information is explained below.

Foundry — The type foundry (in this case, Adobe) that digitized and supplied the font.

*A property is a piece of information associated with a window or a font. See Volume One, *Xlib Programming Manual*, for more information about properties.

Set width — A value describing a font's proportionate width, according to the foundry. Typical set widths include: normal, condensed, narrow, double width. All of the newer fonts have the set width *normal*.

Pixels and points — Type is normally measured in points, a printer's unit equal to 1/72 of an inch. The size of a font in pixels depends on the resolution of the display font in pixels. For example, if the display font has 100 dots per inch (dpi) resolution, a 12 point font will have a pixel size of 17, while with 75 dpi resolution, a 12 point font will have a pixel size of 12.

Spacing — Either m (monospace, i.e., fixed-width) or p (proportional, i.e., variable-width).

Horizontal and vertical resolution — The resolution in dots per inch that a font is designed for. Horizontal and vertical figures are required because a screen may have different capacities for horizontal and vertical resolution.

Average width — Mean width of all characters in the font, measured in tenths of a pixel, in this case 6 pixels.

Character set — ISO, the International Standards Organization, has defined character set standards for various languages. The iso8859-1 in Figure B-2 represents the ISO Latin 1 character set, which is used by all of the fonts in the *75dpi* and *100dpi* directories. The ISO Latin 1 character set is a superset of the standard ASCII character set, which includes various special characters used in European languages other than English. See Appendix H of Volume Two, *Xlib Reference Manual*, for a complete listing of the characters in the ISO Latin 1 character set.

This font-naming convention is intended to allow for the unique naming of fonts of any style, resolution and size. It is powerful, but unwieldy.

To create a label widget that displays text in the font stored in the file *courBO10*, you could use the resource setting:

```
*label:  -adobe-courier-bold-o-normal--10-100-75-75-m-60-iso8859-1
```

Since typing a font name of this length is neither desirable nor practical, the X Window System developers have provided two alternatives: wildcarding and aliasing.

B.2.2 Font Name Wildcarding

Any unnecessary part of a font name can be "wildcarded" by specifying a question mark (*?*) for any single character and an asterisk (*) for any group of characters.

For example, using a wildcarded font name, the resource specification above could be written:

```
*label:  *courier-bold-o-*-100*
```

(Note that when using wildcards with the *-fn* command line option, you must take care to quote the font names, since the UNIX shell has special meanings for the wildcard characters *

and ?. This can be done by enclosing the entire font name in quotes, or by escaping each wildcard character by typing a backslash before it.)

If more than one font in a given directory matches a wildcarded font name, the server chooses the font to use. If fonts from *more than one directory* match the wildcarded name, the server will always choose a font from the directory that is earlier in the font path. Thus, if a wildcarded font name matches a font from both the *75dpi* and *100dpi* directories, and the *75dpi* directory comes first in the font path, the server chooses the font from that directory.

In creating a wildcarded font name, you need to decide which parts of the standard font name must be explicit and which parts can be replaced with wildcards. As the previous example illustrates, you can use a single wildcard character for multiple parts of the font name. For instance, the final asterisk in the example stands for the sequence:

```
-75-75-m-60-iso8859-1
```

in the explicit font name. The idea is to specify enough parts of the font name explicitly so that the server gives you the font you have in mind.

It's helpful to familiarize yourself with the available font families, weights, slants, and point sizes. The following list gives these statistics for the fonts in the directories *75dpi* and *100dpi* in the standard X distribution from MIT.* (The fonts in the *misc* directory are holdovers from Release 2 and have short, manageable names that should not require wildcarding.)

Font families	Charter, Courier, Helvetica, New Century Schoolbook, Symbol, Times
Weights	Medium, bold
Slants	Roman (r), an upright design Italic (i), an italic design slanted clockwise from vertical Oblique (o), an obliqued upright design, slanted clockwise from vertical
Point sizes	8, 10, 12, 14, 18, 24

If you're unfamiliar with the general appearance of a particular font family, try displaying one of the fonts with *xfd*, as described in Volume Three, *X Window System User's Guide*.

As a general rule, we suggest you type the following parts of a font name explicitly:

- Font family
- Weight
- Slant
- Point size

Note that it's better to match the point size field, which is measured in tenths of a point (the 100 in the previous example, equal to 10 points), than the pixel field (the 10). This allows your wildcarded font name to work properly with monitors of different resolutions. For

*For fonts other than those shipped by MIT, other families, weights, slants, point sizes, etc., may apply.

example, say you use the following name to specify a 24 point (size), medium (weight), Italic (slant) Charter (family) font:

```
*charter-medium-i-*-240-*
```

This will match either of the following two font names (the first for 75 dpi monitors and the second for 100 dpi monitors):

```
-bitstream-charter-medium-i-normal--25-240-75-75-p-136-iso8859-1
-bitstream-charter-medium-i-normal--33-240-100-100-p-136-iso8859-1
```

depending on which directory comes first in your font path. Specifying font size explicitly in pixels (25 for the first or 33 for the second) rather than in points would limit you to matching only one of these fonts.

Given the complexity of font names and the rules of precedence used by the server, you should use wildcards carefully.

B.2.3 Font Name Aliasing

Another way to abbreviate font names is by aliasing—that is, by associating them with alternative names. You can create a file (or files) called *fonts.alias*, in any directory in the font search path, to set aliases for the fonts in that directory. The X server uses both *fonts.dir* files (see Section C.2.5) and *fonts.alias* files to locate fonts in the font path.

Be aware that when you create or edit a *fonts.alias* file, the server does not *automatically* recognize the aliases in question. You must make the server aware of newly created or edited alias files by resetting the font path with *xset* as described in Section C.2.4.

The *fonts.alias* file has a two-column format similar to the *fonts.dir* file (described in a moment): the first column contains aliases, the second contains the actual font names. If you want to specify an alias that contains spaces, enclose the alias in double quotes. If you want to include double quotes or other special characters as part of an alias, precede each special symbol with a backslash.

When you use an alias to specify a font in a command line, the server searches for the font associated with that alias in every directory in the font path. Therefore, a *fonts.alias* file in one directory can set aliases for fonts in other directories as well. You might choose to create a single alias file in one directory of the font path to set aliases for the most commonly used fonts in all the directories. Example B-1 shows a sample *fonts.alias* file.

Example B-1. Sample fonts.alias file

```
xterm12      -adobe-courier-medium-r-normal--12-120-75-75-m-70-iso8859-1
xterm14      -adobe-courier-medium-r-normal--14-140-75-75-m-90-iso8859-1
xterm18      -adobe-courier-medium-r-normal--18-180-75-75-m-110-iso8859-1
```

As the names of the aliases suggest, this sample file contains aliases for three fonts (of different point sizes) that are easily readable in *xterm* windows.

You can also use wildcards within the font names in the right hand column of an alias file. For instance, the alias file above might also be written:

```
xterm12    *courier-medium-r-*-120*
```

```
xterm14    *courier-medium-r-*-140*
xterm18    *courier-medium-r-*-180*
```

Once the server is made aware of aliases, you can specify an alias in resource specifications or on the command line:

```
xterm.font: xterm12
```

or:

```
% xterm -fn xterm12
```

If you are accustomed to the Release 2 font naming convention (each font name being equivalent to the name of the file in which it is stored, without the *.snf* extension), there is a way to emulate this convention using alias files. In each directory in the font path, create a *fonts.alias* file containing only the following line:

```
FILE_NAMES_ALIASES
```

Each filename (without the *.snf* extension) will then serve as an alias for the font the file contains. Note that an alias file containing this line applies only to the directory in which it is found. To make every font name equivalent to the name of the file in which it is stored, you need to create a *fonts.alias* file such as this in every font directory.

If you've specified FILE_NAMES_ALIASES in an alias file, you can choose the fonts in that directory by means of their filenames, as we did in the resource example at the end of Chapter 2, *Introduction to the X Toolkit*.

B.2.4 Making the Server Aware of Aliases

After you create (or update) an alias file, the server does not automatically recognize the aliases in question. You must make the server aware of newly created or edited alias files by "rehashing" the font path with *xset*. Enter:

```
% xset fp rehash
```

on the command line. The *xset* option *fp* (font path) with the `rehash` argument causes the server to reread the *fonts.dir* and *fonts.alias* files in the current font path. You need to do this every time you edit an alias file. (You also need to use *xset* if you add or remove fonts. See Volume Three, *X Window System User's Guide*, for details.)

B.2.5 The fonts.dir Files

In addition to font files, each font directory contains a file called *fonts.dir*. The *fonts.dir* files serve, in effect, as databases for the X server. When the X server searches the directories in the default font path, it uses the *fonts.dir* files to locate the font(s) it needs.

Each *fonts.dir* file contains a list of all the font files in the directory with their associated font names, in two-column form. (The first column lists the font file and the second column lists the actual font name associated with the file.) The first line in *fonts.dir* lists the number of entries in the file (i.e., the number of fonts in the directory).

Example B-2 shows the *fonts.dir* file from the directory */usr/lib/X11/fonts/100dpi*. As the first line indicates, the directory contains 24 fonts.

Example B-2. fonts.dir file in /usr/lib/X11/fonts/100dpi

```
24
charBI08.snf -bitstream-charter-bold-i-normal--11-80-100-100-p-68-iso8859-1
charBI10.snf -bitstream-charter-bold-i-normal--14-100-100-100-p-86-iso8859-1
charBI12.snf -bitstream-charter-bold-i-normal--17-120-100-100-p-105-iso8859-1
charBI14.snf -bitstream-charter-bold-i-normal--19-140-100-100-p-117-iso8859-1
charBI18.snf -bitstream-charter-bold-i-normal--25-180-100-100-p-154-iso8859-1
charBI24.snf -bitstream-charter-bold-i-normal--33-240-100-100-p-203-iso8859-1
charB08.snf -bitstream-charter-bold-r-normal--11-80-100-100-p-69-iso8859-1
charB10.snf -bitstream-charter-bold-r-normal--14-100-100-100-p-88-iso8859-1
charB12.snf -bitstream-charter-bold-r-normal--17-120-100-100-p-107-iso8859-1
charB14.snf -bitstream-charter-bold-r-normal--19-140-100-100-p-119-iso8859-1
charB18.snf -bitstream-charter-bold-r-normal--25-180-100-100-p-157-iso8859-1
charB24.snf -bitstream-charter-bold-r-normal--33-240-100-100-p-206-iso8859-1
charI08.snf -bitstream-charter-medium-i-normal--11-80-100-100-p-60-iso8859-1
charI10.snf -bitstream-charter-medium-i-normal--14-100-100-100-p-76-iso8859-1
charI12.snf -bitstream-charter-medium-i-normal--17-120-100-100-p-92-iso8859-1
charI14.snf -bitstream-charter-medium-i-normal--19-140-100-100-p-103-iso8859-1
charI18.snf -bitstream-charter-medium-i-normal--25-180-100-100-p-136-iso8859-1
charI24.snf -bitstream-charter-medium-i-normal--33-240-100-100-p-179-iso8859-1
charR08.snf -bitstream-charter-medium-r-normal--11-80-100-100-p-61-iso8859-1
charR10.snf -bitstream-charter-medium-r-normal--14-100-100-100-p-78-iso8859-1
charR12.snf -bitstream-charter-medium-r-normal--17-120-100-100-p-95-iso8859-1
charR14.snf -bitstream-charter-medium-r-normal--19-140-100-100-p-106-iso8859-1
charR18.snf -bitstream-charter-medium-r-normal--25-180-100-100-p-139-iso8859-1
charR24.snf -bitstream-charter-medium-r-normal--33-240-100-100-p-183-iso8859-1
```

The *fonts.dir* files are created by the *mkfontdir* client when X is installed. *mkfontdir* reads the font files in directories in the font path, extracts the font names, and creates a *fonts.dir* file in each directory. If *fonts.dir* files are present on your system, you probably won't have to deal with them, or with *mkfontdir*, at all. If the files are not present, or if you have to load new fonts or remove existing ones, you will have to create files with *mkfontdir*. Refer to Volume Three, *X Window System User's Guide*, for details.

B.3 Window Geometry

All clients that display in a window take a geometry option that specifies the size and location of the client window.* The syntax of the geometry option is:

```
-geometry geometry
```

The *-geometry* option can be (and often is) abbreviated to *-g*, unless there is a conflicting option that begins with *g*.

*The Release 2 syntax:

```
=geometry
```

is no longer supported.

The corresponding resource is `XtNGeometry`, which can be set in a resource file as follows:

```
*mywidget.geometry:  geometry_string
```

The argument to the geometry option (*geometry*), referred to as a "standard geometry string," has the form:

```
widthxheight±xoff±yoff
```

The variables *width* and *height*, are values in pixels for many clients. However, application developers are encouraged to use units that are meaningful to the application. For example, *xterm* uses columns and rows of text as width and height values in the *xterm* window. *xoff* (x offset), and *yoff* (y offset) are always in pixels.

You can specify any or all elements of the geometry string. Incomplete geometry specifications are compared to the resource manager defaults and missing elements are supplied by the values specified there. If no default is specified there, and *uwm* is running, the window manager will require you to place the window interactively.

The values for the *x* and *y* offsets have the following effects:

Table B-1. Geometry Specification: x and y Offsets

Offset Variables	Description
+xoff	A positive *x* offset specifies the distance that the left edge of the window is offset from the left side of the display.
+yoff	A positive *y* offset specifies the distance that the top edge of the window is offset from the top of the display.
-xoff	A negative *x* offset specifies the distance that the right edge of the window is offset from the right side of the display.
-yoff	A negative *y* offset specifies the distance that the bottom edge of the window is offset from the bottom of the display.

For example, the command line:

```
% xclock -geometry 125x125-10+10 &
```

places a clock 125x125 pixels in the upper-right corner of the display, 10 pixels from both the top and the right edge of the screen.

For *xterm*, the size of the window is measured in characters and lines. (80 characters wide by 24 lines long is the default terminal size.) If you wanted to use the vt100 window in 132-column mode, with 40 lines displayed at a time, you could use the following geometry options:

```
% xterm -geometry 132x40-10+350 &
```

This will place an *xterm* window 132 characters wide by 40 lines long in the lower-right corner, 10 pixels from the right edge of the screen and 350 pixels from the top of the screen.

Some clients may allow you to specify geometry strings for the size and position of the icon or an alternate window, usually through resources (in an *Xdefaults* or other resource file). See the appropriate client reference pages in Part Three of Volume Three, *X Window System User's Guide*, for a complete list of available resources.

You should be aware that, as with all user preferences, you may not always get exactly what you ask for. Clients are designed to work with a window manager, which may have its own rules for window or icon size and placement. However, priority is always given to specific user requests, so you won't often be surprised.

C

Naming Conventions

This appendix describes a suggested set of conventions for naming widgets, and elements within widget code.

C
Naming Conventions

This appendix proposes a set of conventions for naming certain elements of widget code.* If the naming conventions used in all widget sets are consistent, there will be several benefits:

- It will be much easier for programmers to move from toolkit to toolkit without needing to constantly refer to manuals to figure out how to properly name various items.
- It will be easier to mix widgets from different widgets sets in one application.
- It will make it possible for automatic code generators to work with lots of widgets without making special modifications for any toolkit.

These conventions are common between the OPEN LOOK and Motif widget sets with a few minor differences. However, the Athena widget set currently does not follow the conventions described.

It is important to note that these suggestions are in no way blessed (or damned) by the X Consortium—they are simply a guideline that we suggest you follow in the interests of promoting the benefits listed above. We will use Motif as an example.

A toolkit uses a special prefix with all its widgets. In the case of Motif, this prefix is *m*. Using that prefix and a hypothetical Label widget as an example, the conventions are as follows:

Toolkit library name	*libXm.a*
Widget class name	`XmLabel`
Include directives	`<Xm/Label.h>`
`class_name` field in the core structure	`Label`
Enumerated resource values	`XmCAPITALIZED_WORDS`
Public function names	`XmLabelFunctionName()`

*This appendix is based on a *comp.windows.x* network news posting by Gene Dykes of Cornell University.

Truncated include file names	Strip the lower case letters a word at a time until the basename is nine characters or fewer (but strip as few letters as possible from the initial word). Thus:
	`<Xm/VeryLongWidgetName.h>`
	becomes:
	`<Xm/VeryLWN.h>`
	but:
	`<Xm/Verylongwidgetname.h>`
	becomes:
	`<Xm/Verylongw.h>`
	(Note difference in `VeryLong` (two words) and `Verylong` (one word).)

Macro names for preprocessor in include files	`#ifndef XM_LABEL_H` This is to prevent header files from being included more than once.
	If a widget has a corresponding gadget, then `Gadget` is appended to the widget name, so `XmLabelGadget` is the gadget class name.
Widget class pointer	`xmLabelWidgetClass`
Gadget class pointer	`xmLabelGadgetClass`
Create function for widgets or gadgets	`Widget XmCreateLabel(parent,name, arglist,argcount)` This is a shortcut to using `XtCreateWidget`. However, in the case of top-level widgets (menus, dialogs, MainWindow), it also creates the shell widget and creates the requested widget within it.
	The special Create functions can also be used as convenience routines for specialized widget instances. For example, `XmCreateWorkingDialog` and `XmCreateWarningDialog` actually create a MessageBox whose `XmDialogType` resources are respectively `XmDIALOG_WARNING` and `XmDIALOG_WORKING`.

OPEN LOOK uses conventions similar to Motif, but leaves the prefixes off the widget class name. We recommend that you supply a prefix for any widgets you write, especially if there is any chance that some other widget set may use the same class name. In other words, if we wrote a new Label widget, we might call it OraLabel, not just Label, so that we could still use the Athena Label widget in the same application.

D

Release Notes

This appendix summarizes the changes between Release 3 and Release 4 of X Toolkit, as concerns writing or converting widgets and applications.

In This Chapter:

Summary of New Functions and Procedure Types 467
Application Writing 470
New Function for Initializing an Application 470
Varargs Interfaces 470
Special Types for Varargs List Members 471
Loading the Resource Database 471
The Language String 471
New Built-In Application Resources 472
Fallback Resources 472
Parsing the Command Line 472
Changes Affecting Both Application and Widget Writing 473
Actions 473
Directly Invoking Actions 473
Action Hook Registration 473
Obtaining Event Information in an Action Procedure 473
Registering Button and Key Grabs For Actions 474
Action Functions and Unrealized Widgets 474
Reserved Names for Resources and Actions 474
Resource String Identifier Convention 474
Resource File Format 474
Modifier Names 475
Determining Specification Revision Level 475
Implementation-specific Datatype Definitions 475
Event Handlers 476
Error and Warning Handlers 476
Changes Affecting Widget Writing 476
New Name for Core Class 476

File Content Changes 477
Changes to Private .h Files 477
Changes to .c Files 477
Class Extension Structures 477
Composite Class Extension Structure 478
Constraint Class Extension Structure 479
New Constraint get_values_hook method 479
Shell Class Extension Structure 479
New XtNvisual Resource 480
Methods 480
Mapping Order of Realized Child Widgets 481
Callback Lists 481
XtNunrealizeCallback 481
Internal Format of Callback Lists 481
New Routine to Call Callback List 482
Resource Lists 482
Base Offsets in Resource List 482
XtProcedureArg 482
Default Resource Values 482
Getting a Constraint Resource List 482
Converters 482
New Xt Resource Representations 483
New Xt Resource Converters 483
New Fn for Issuing Conversion Warnings 484
New Resource Converters Interface 484
Keyboard Handling 486
KeyCode-to-KeySym Conversions 486
KeySym-to-KeyCode Conversions 486
Selections 486
Objects 486
The Gadget Parent 487
Xt Functions that take any Object 487
Xt Functions that take any RectObj 489
Xt Functions that take any Core 489
Xt Functions that take any Composite 490
Xt Functions that take any Shell or a Subclass 490
Macros for Getting Object Information 490
Miscellaneous 490
XtNameToWidget 490
New Exposure Compression Controls 491
Modal Cascades 492
Requesting Key and Button Grabs 492
New Macros for Widget Superclass Determination 493
Macros and Procedure Equivalents 493
Finding File Names 493
Improved Multi-Display Support in Xmu 493

D
Release Notes

The Xt specification is the definition of the X Consortium standard for Xt. This appendix describes the changes to the Xt Intrinsics specification between Release 3 and Release 4. This appendix separates them according to whether they affect just application writers, just widget writers, or both, so that you can read the sections appropriate to your needs.

In the majority of cases, Xt applications correctly written for Release 3 will be source compatible with Release 4. However, there have been many additions to Xt that you may wish to use when converting an application or widget to R4 or writing one from scratch. This is a fairly complete summary of the additions and changes that have been made to the Intrinsics since Release 3.

Some of the information provided below describes behavior that is not new but that is newly documented in R4—behavior that was in the R3 implementation of Xt (and still present in R4) but was not in the R3 Xt specification. These places are noted.

D.1 Summary of New Functions and Procedure Types

`XtAppInitialize` is the new function to be used for initializing Xt applications.

`XtAppSetFallbackResources` allows you to provide a set of backup resources that will be used in case the app-defaults file is not installed. An argument in `XtAppInitialize` is provided for this purpose, so `XtAppSetFallbackResources` is rarely needed.

The following functions are the new varargs versions of existing functions: `XtVaSetValues`, `XtVaGetValues`, `XtVaCreateWidget`, `XtVaCreateManagedWidget`, `XtVaAppCreateShell`, `XtVaGetSubresources`, `XtVaGetApplicationResources`, `XtVaCreatePopupShell`, `XtVaSetSubvalues`, `XtVaGetSubvalues`, and `XtVaAppInitialize`.

`XtInitializeWidgetClass` initializes a widget or object class without creating any widget or object. This is normally done in order to make available the type converters registered by that class before the first instance of that class is created, so that resources for the first instance can be set while creating it using `XtVaTypedArg`, a new feature of `XtVaSetValues` and `XtVaCreateWidget`.

XtVaCreateArgsList is for copying an argument list before using the XtVaNestedList symbol in one of the varargs interfaces.

XtPopupSpringLoaded pops up a widget and creates a global passive grab of the button being held.

The following functions are new Xt interfaces for making global passive and active grabs: XtGrabKey, XtUngrabKey, XtGrabKeyboard, XtUngrabKeyboard, XtGrabButton, XtUngrabButton, XtGrabPointer, and XtUngrabPointer. These are rarely needed since the grabbing needed to implement pop ups is built in to existing functions such as XtMenuPopup and the new function XtPopupSpringLoaded.

XtInsertEventHandler and XtInsertRawEventHandler are new functions that allow you to position an event handler at the beginning or end of the current list of event handlers for a certain event. They are also capable of moving an already registered event handler in the list.

XtCallCallbackList is similar to XtCallCallbacks but is slightly more efficient because it avoids the lookup of the callback resource in the widget.

XtGetConstraintResourceList is the new analogue to the existing XtGetResourceList. Both functions give you access to a widget's resource list.

XtDisplayStringConversionWarning supercedes XtStringConversionWarning for use in type converters that convert from XtRString.

R4 provides a completely new interface for type converters, which provides more controllable caching and display-specific conversions. The new function for registering a type converter is XtAppSetTypeConverter. There is a new type converter calling sequence, described in XtConverter(2) in Volume Five, *X Toolkit Intrinsics Reference Manual*. The new functions for calling a converter directly are XtConvertAndStore, XtCallConverter, though these are less necessary now that the new XtVaTypedArg symbol in the varargs interfaces is available.

XtOffsetOf is a new, more portable version of XtOffset (used in defining a resource list). Also in the resource list, the symbol XtProcedureArg, when used in the field normally specified as XtBaseOffset, instructs Xt to get the default value by calling the function of type XtConvertArgProc that is specified in the default value field.

The new functions for controlling the caching of type converters are XtCallbackReleaseCacheRef, XtCallbackReleaseCacheRefList, and XtAppReleaseCacheRefs.

Xt now provides an action hook, which allows you to record the actions being invoked in an application for later playback (this is one way to implement keyboard macros). XtAppAddActionHook registers a function that will be called whenever any action in the application is invoked. The action hook function is passed all the information passed to the action, plus the string action name. XtRemoveActionHook lets you unregister an action hook function. XtCallActionProc lets you call an action function directly to play back recorded actions. The function prototype of an action hook is XtActionHookProc.

`XtGetActionKeysym` is used within an action routine to get the keysym that invoked the action. This is useful because the event passed to the action contains only the keycode.

`XtGetKeysymTable` and `XtKeysymToKeycodeList` are two more functions for controlling the mapping of keycodes to keysyms. These functions are rarely needed.

`XtRegisterGrabAction` allows you to write an action that when invoked will automatically result in a global grab of the pointer or key that invoked the action. Internally in Xt this is used to register `XtMenuPopup`, so it is rarely needed.

`XtLastTimestampProcessed` is useful in selection processing, since some events that trigger selections do not include a timestamp. `XtGetSelectionRequest` is used in the *`convert_proc`* in selections to get the time from the selection request event.

`XtDisplayOfObject`, `XtScreenOfObject`, and `XtWindowOfObject` are the equivalents of `XtDisplay`, `XtScreen`, and `XtWindow` that work for objects, which are the windowless and sometimes geometryless forms of widgets. `XtIsObject` is a macro that lets you distinguish an object from a widget.

`XtDisplayToApplicationContext` is sometimes useful in applications that open connections to more than one display.

`XtName` converts a widget ID to a widget name, the opposite of `XtNameToWidget`.

`XtSetWMColormapWindows` sets the value of the WM_COLORMAP_WINDOWS property on a widget's window. This is done in order to tell the window manager that windows in this application other than the top-level window need special colormaps installed.

Release 4 introduces a set of new selection interfaces which allow the owner and requestor to handle the selection in arbitrary chunks. This is called an *incremental* transfer, to differentiate it from the existing interfaces that implement an *atomic* transfer. The protocols for both are very similar, but the procedures written by the owner and requestor to convert and receive data may be called multiple times in an incremental transfer, instead of just once as in the atomic transfer. The new functions and procedure types are: `XtCancelConvertSelectionProc`, `XtCancelSelectionCallbackProc`, `XtConvertSelectionIncrProc`, `XtLoseSelectionIncrProc`, `XtSelectionDoneIncrProc`, `XtSelectionIncrCallbackProc`, `XtGetSelectionValueIncremental`, `XtGetSelectionValuesIncremental`, and `XtOwnSelectionIncremental`. See Chapter 10, *Interclient Communications*, for a description of how these functions and procedure types are used.

`XtGetApplicationNameAndClass` allows an application to get its own resource name and class. This function is rarely needed.

Xt implements its handling of the language string using `XtResolvePathname`, which calls `XtFindFile`. These two functions are provided for applications that need to search the file system for files. `XtResolvePathname` searches using X/Open language localization conventions, while `XtFindFile` searches arbitrary paths.

D.2 Application Writing

The major changes from an application writer's point of view include windowless objects, varargs-style interfaces, fallback resources, and locale-driven finding of data files.

D.2.1 New Function for Initializing an Application

Instead of `XtInitialize`, applications should now use `XtAppInitialize` or `XtVaAppInitialize` to initialize the X Toolkit internals, create an application context, open and initialize a display and create the initial application shell instance. Its arguments are somewhat different from `XtInitialize`. See Chapter 3, *More Techniques for Using Widgets*, or Volume Five, *X Toolkit Intrinsics Reference Manual*, for details. `XtAppInitialize` calls `XtToolkitInitialize`, `XtCreateApplicationContext`, `XtAppSetFallbackResources`, `XtOpenDisplay`, and `XtAppCreateShell`.

One argument of `XtAppInitialize` is a string containing resources that will be the backup in case the application's app-defaults file is not present. These are called fallback resources, and the argument is *`fallback_resources`*.

D.2.2 Varargs Interfaces

All Xt interfaces that require `ArgList` arguments now have analogs which conform to the ANSI C variable-length argument list ("varargs") calling convention. The name of the analog is formed by prefixing "Va" to the name of the corresponding `ArgList` procedure. For example, the varargs version of `XtCreateWidget` is `XtVaCreateWidget`. Each procedure whose name starts with `XtVa` takes as its last arguments, in place of the corresponding `ArgList/Cardinal` parameters, a variable-length parameter list of resource name and value pairs where each name is of type `String` and each value is of type `XtArgVal`. The end of the list is identified by a name entry containing `NULL`. The `ArgList` and varargs interfaces may be used interchangeably within the same application. Table D-1 shows the `ArgList` functions and the analogous varargs functions:

Table D-1. ArgList Functions and Varargs Counterparts

ArgList	Varargs
`XtAppCreateShell`	`XtVaAppCreateShell`
`XtAppInitialize`	`XtVaAppInitialize`
`XtCreatePopupShell`	`XtVaCreatePopupShell`
`XtCreateManagedWidget`	`XtVaCreateManagedWidget`
`XtCreateWidget`	`XtVaCreateWidget`
`XtGetApplicationResources`	`XtVaGetApplicationResources`
`XtGetSubresources`	`XtVaGetSubresources.`
`XtGetSubvalues`	`XtVaGetSubvalues`
`XtGetValues`	`XtVaGetValues`

Table D-1. ArgList Functions and Varargs Counterparts (continued)

ArgList	Varargs
`XtSetSubvalues`	`XtVaSetSubvalues`
`XtSetValues`	`XtVaSetValues`

D.2.2.1 Special Types for Varargs List Members

Two special symbols are defined for use only in varargs lists: `XtVaTypedArg` and `XtVaNestedList`. Each can be used in place of a resource name to indicate that the following fields should be treated differently than a normal resource value. `XtVaTypedArg` allows you to use the subsequent four arguments to instruct Xt how to invoke the proper type converter to convert the given value into the type required by the widget. To use `XtVaTypedArg` while creating the first instance of a widget class, however, you must call `XtInitializeWidgetClass` first.

`XtVaNestedList` lets you include an `ArgList` in a varargs list. This is normally done to include the same list of resource settings in several calls to create widgets. `XtVaCreateArgsList` dynamically allocates a varargs list, for use with `XtVaNestedList` in multiple calls.

For more information on `XtVaTypedArg` and `XtVaNestedList`, see Chapter 3, *More Techniques for Using Widgets*.

D.2.3 Loading the Resource Database

There have been several extensions in the way the resource database is loaded.

D.2.3.1 The Language String

In R4, a mechanism for making applications portable between languages is introduced. It allows the application writer to create separate app-defaults files for each language. Each of these files will have the same name, but they will have different paths. A *language string* determines the path.* In other words, Xt will look for the app-defaults file in */usr/lib/X11/LANG/app-defaults* before it looks in */usr/lib/X11/app-defaults*. This is done for both the application's app-defaults file and the user's XAPPLRESDIR file. For details on the exact order of searching and merging for resource files, see Chapter 9, *Resource Management and Type Conversion*.

*This mechanism conforms to the *X/Open Portability Guide* language localization conventions.

The language string is a user-specified value. The language string can be set by setting the LANG environment variable, by setting the -xnlLanguage command line option, or by setting the "xnlLanguage" resource in a resource file. If no language string is found, the empty string is used, and Xt defaults to its R3 behavior.

The environment variable XUSERFILESEARCHPATH can be set to define a path that will be searched before the directory defined by XAPPLRESDIR.

D.2.3.2 New Built-In Application Resources

There are three new built-in application resources that affect the global operation of Xt.

`XtNxnlLanguage`, class `XtCXnlLanguage` sets the language string to be used when reading application and user resource files.

`XtNDefaultFont`, class `XtCDefaultFont` allows the application to supply a font name to use if the font supplied for any other resource cannot be loaded.

`XtNmultiClickTime`, class `XtCMultiClickTime` sets the time used by Xt to detect double and triple clicks in the translation manager. This resource may also be modified dynamically by the application, which is useful since the multi-click time is unique for each Display. If no value is specified, the initial value is 200 milliseconds.

`XtSetMultiClickTime` is a convenience routine to set the multi-click time dynamically, and `XtGetMultiClickTime` reads the multi-click time.

D.2.3.3 Fallback Resources

One argument of `XtAppInitialize`, *`fallback_resources`*, allows you to specify backup resource settings in case the app-defaults file is not installed properly. You can set a complete copy of the app-defaults file here, or a minimal set of resources that makes the application tell the user that the real app-defaults file in not installed. For an example of setting fallback resources, see Chapter 9, *Resource Management and Type Conversion*.

The fallback resources can also be set with `XtAppSetFallbackResources`, although this is rarely needed since an argument is provided in `XtAppInitialize`.

D.2.4 Parsing the Command Line

There are two new option styles that you can specify in an option list so that Xt will parse them. They are `XrmoptionResArg`, which instructs Xt to use the next argument as input to `XrmPutLineResource`, and `XrmoptionSkipNArgs`, which instructs Xt to ignore this option and the next `OptionDescRec.value` arguments. These option styles are not usually needed. For more on defining option styles, see Chapter 3, *More Techniques for Using Widgets*.

The RESOURCE_NAME environment variable is now used. If no -name option is specified on the command line when the user invokes the application, `XtAppInitialize` (or `XtOpenDisplay` if called directly) reads the RESOURCE_NAME environment variable to get

an application name to be used in extracting applicable resource specifications from the resource database. If RESOURCE_NAME is not set, the final component of `argv[0]` is used. If `argv[0]` does not exist or is the empty string, the application name is "main".

D.3 Changes Affecting Both Application and Widget Writing

D.3.1 Actions

Several new functions associated with actions have been defined.

D.3.1.1 Directly Invoking Actions

Normally, action procedures are invoked by the Intrinsics when an event or event sequence arrives for a widget. Occasionally it may be desirable to invoke an action procedure directly, without generating (or synthesizing) an event. One such circumstance is an application or subclass action routine which envelopes an existing (superclass) action routine. `XtCallActionProc` can be used to do this. `XtCallActionProc` is also useful for playing back actions recorded with `XtAppAddActionHook`.

D.3.1.2 Action Hook Registration

An application can use `XtAppAddActionHook` to specify a procedure that will be called just before every action routine is dispatched by the translation manager.

Action hooks should not modify any of the data pointed to by the arguments other than the *client_data* argument.

`XtRemoveActionHook` removes an action hook procedure. (Notice that the `XtAppAddActionHook` takes an application context argument, which `XtRemoveActionHook` doesn't, thus the asymmetry in their names.)

D.3.1.3 Obtaining Event Information in an Action Procedure

When an action procedure is invoked on a `KeyPress` or `KeyRelease` event, it often needs the keysym and modifiers corresponding to the event which caused it to be invoked, since the event passed in the action procedure contains only the keycode. In order to avoid repeating the processing that was just performed by the Intrinsics to match the translation entry, the keysym and modifiers are stored for the duration of the action procedure and are made available to the client with `XtGetActionKeysym`. `XtGetActionKeysym` retrieves the keysym and modifiers that matched the final event specification in the translation table entry.

D.3.1.4 Registering Button and Key Grabs For Actions

`XtRegisterGrabAction` registers a grab that will be initiated when the given action is invoked in response to `KeyPress` or `ButtonPress` events. This function is used internally by Xt to register the grab for the `XtMenuPopup` action.

D.3.1.5 Action Functions and Unrealized Widgets

Action procedures may not assume that the widget in which they are invoked is realized; an accelerator specification can cause an action procedure to be called for a widget which does not yet have a window. Widget writers should note which of a widget's callback lists are invoked from action procedures and warn clients not to assume the widget is realized in those callbacks. (This is newly documented but also true of R3.)

D.3.2 Reserved Names for Resources and Actions

Resource names beginning with the two-character sequence "xt" and resource classes beginning with the two-character sequence "Xt" are reserved to the Intrinsics for implementation-dependent uses. Consequently, all symbols beginning with `XtNxt` and `XtCXt` are reserved.

The Intrinsics reserve all action names and parameters starting with the characters "Xt" for future standard enhancements. Users, application and widgets should not declare action names or pass parameters starting with these characters except to invoke specified built-in functions.

In general, applications should not declare or define anything beginning with Xt in any case, since these are generally associated with the Xt standard and could conflict or be confusing.

D.3.3 Resource String Identifier Convention

Definitions of XtN, XtC, and XtR constants should have just a single space between the symbol and the value, and no comment. This prevents preprocessor warnings from multiple, slightly different definitions for the same symbol.

D.3.4 Resource File Format

As of R4, the resource file format is defined in the Xlib specification instead of by Xt, and the following was added: To include arbitrary octets in a string, use the 4-character sequence "\nnn" where nnn is the numeric value of the octet specified as an octal constant. For example, a value containing a `NULL` byte may be stored by including "\000" in the string. This is one way to specify ISO-Latin1 strings that are above the 128 ASCII characters—these strings are needed for Western languages other than English.

D.3.5 Modifier Names

If both an exclamation point (!) and a colon (:) are specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and that no other modifiers except the standard modifiers can be asserted. Any standard modifiers in the event are applied as for colon (:) above.

D.3.6 Determining Specification Revision Level

Widget and application developers who wish to maintain a common source pool which will build properly with implementations of the Intrinsics at different revision levels of the Xt specification may use the symbolic macro `XtSpecificationRelease`.

Since `XtSpecificationRelease` is new to Release 4, widgets and applications desiring to build against earlier implementations should test for the presence of this symbol and assume only Release 3 interfaces if the definition is not present.

D.3.7 Implementation-specific Datatype Definitions

To increase the portability of widget and application source code between different system environments, Xt defines several datatypes whose precise representation is explicitly dependent upon, and chosen by, each individual implementation. For best portability, applications must only depend on the data type characteristics listed here. These characteristics were not documented until Release 4.

These implementation-defined datatypes are:

`Boolean` — A datum that contains a zero or non-zero value. Unless explicitly stated, clients should not assume that the non-zero value is equal to the symbolic value `True`.

`Cardinal` — An unsigned datum with a minimum range of [0..2^16-1].

`Dimension` — An unsigned datum with a minimum range of [0..2^16-1].

`Position` — A signed datum with a minimum range of [-2^15..2^15-1].

`XtPointer` — A datum large enough to containa `char*`, `int*`, function pointer, structure pointer, or `long` value. A pointer to any type or function, or a `long` value, may be converted to an `XtPointer` and back again and the result will compare equally to the original value. In ANSI C environments it is expected that `XtPointer` will be defined as `void*`.

`XtArgVal` — A datum large enough to contain an `XtPointer`, `Cardinal`, `Dimension`, or `Position` value.

`XtEnum` — A datum large enough to encode at least 128 distinct values, two of which are the symbolic values `True` and `False`. The symbolic values `TRUE` and `FALSE` are also defined to be equal to `True` and `False`, respectively.

D.3.8 Event Handlers

Event handlers now have an additional final argument, *continue_to_dispatch*. This argument specifies whether or not the remaining event handlers registered for the current event should be called. This feature should be used with care.

`XtInsertEventHandler` registers an event handler procedure before or after all previously registered event handlers.

`XtInsertRawEventHandler` registers an event handler procedure that receives events before or after all previously registered event handlers without selecting for the events.

D.3.9 Error and Warning Handlers

The routines that install error and warning handlers now return the previous handlers.

D.4 Changes Affecting Widget Writing

The Xt specification has undergone several sets of revisions in the course of adoption as an X Consortium standard specification. The Release 3 specification was the first X Consortium standard for Xt. Release 4 and future releases will have complete source compatibility with widgets and applications written to previous Consortium standard revisions.

The Intrinsics do not place any special requirement on widget programmers to retain source or binary compatibility for their widgets as they evolve, but several conventions have been established to assist those developers who want to provide such compatibility. In particular, widget programmers may wish to define class extension records, as described in Section 13.13.

D.4.1 New Name for Core Class

Prior to R4, the official name of the Core class was `WidgetClass`. An alias has been added to allow the Core widget to be referenced using the name `CoreWidgetClass`, which makes its naming parallel with all other classes. Similarly, all other symbols that reference Core class structures also have analogous aliases, as shown in Table D-2.

Table D-2. New Naming for Core Class Symbols and Variables

Existing Symbol	Added Symbol
Core Class Structure	
`WidgetClassRec`	`CoreClassRec`
`*WidgetClass`	`*CoreWidgetClass`
`widgetClassRec`	`coreClassRec`
`widgetClass`	`coreWidgetClass`

Table D-2. New Naming for Core Class Symbols and Variables (continued)

Existing Symbol	Added Symbol
Core Instance Structure	
`WidgetRec`	`CoreRec`
`*Widget`	`*CoreWidget`

D.4.2 File Content Changes

D.4.2.1 Changes to Private .h Files

This file will contain the complete widget class extension structure for this widget, if any, and the symbolic constant identifying the class extension version, if any. These extensions are necessary only if the widget writer wants to maintain binary compatibility with previous versions of the widget. This binary compatibility matters most for Xt's own built-in classes. You are unlikely to need to define an extension structure. However, you will need to initialize the extension structures defined by Composite and Constraint when writing subclasses of those classes. See Section 13.13.

D.4.2.2 Changes to .c Files

The `XtVersion` symbol has now been defined as meaning the *implementation* version, as opposed to the specification version. Therefore, code that sets the `version` field of the class structure initialization to `XtVersion` may be rejected at runtime if linked with other implementations. If a widget binary is compatible with other implementations of Xt, the special value `XtVersionDontCheck` should be used in the `version` field to disable version checking. If a widget needs to compile alternative code for different revisions of the Intrinsics interface definition, it may use the symbol `XtSpecificationRelease`.

D.4.3 Class Extension Structures

When a widget class is rewritten, it may be necessary to add new fields to the existing widget class structure. To allow this to be done without requiring recompilation of all subclasses, the last field in a class part record can be an extension pointer. If no extension fields for a class have yet been defined, implementations should set the value of the extension pointer to `NULL`.

If extension fields exist, as is the case with the Composite, Constaint, and Shell classes, subclasses can provide values for these fields by setting the extension pointer in their class structure to point to a statically declared extension record containing the values. Setting the extension field is never mandatory; code that uses fields in the extension record must always check the extension field and do some appropriate default action if it is `NULL`.

In order to permit multiple subclasses and libraries to chain extension records from a single extension field, extension records should be declared as a linked list and each extension record definition should contain the following four fields at the beginning of the structure declaration:

```
struct {
    XtPointer next_extension;
    XrmQuark record_type;
    long version;
    Cardinal record_size;
     ...
};
```

next_extensionSpecifies the next record in the list, or NULL.

record_type Specifies the particular structure declaration to which each extension record instance conforms. The record_type field identifies the contents of the extension record and is used by the allocator of the record to locate its particular extension record in the list. record_type is normally assigned the result of XrmStringToQuark for a registered string constant. The Intrinsics reserve all record type string names beginning with the two characters "XT" for future uses. The value NULLQUARK may also be used by the class part owner in extension records attached to the class part extension field to identify the extension record unique to that particular class.

version Specifies a symbolic constant supplied by the person who defined the structure. The version field is an owner-assigned constant which may be used to identify binary files which have been compiled with alternate definitions of the extension record data structure. The private header file for a widget class should provide a symbolic constant for subclasses to use to initialize this field.

record_size Specifies the total number of bytes allocated for the extension record. The record_size field includes the four common header fields and should normally be initialized with sizeof().

D.4.3.1 Composite Class Extension Structure

A Composite class extension record has been defined. The first four fields in this structure are described above.

```
typedef struct {
    XtPointer next_extension;
    XrmQuark record_type;
    long version;
    Cardinal record_size;
    Boolean  accepts_objects;
} CompositeClassExtensionRec, *CompositeClassExtension;
```

The version field can be initialized using the XtCompositeExtensionVersion symbolic constant.

The accepts_objects field allows Xt to check to make sure that the composite parent is capable of managing the type of child being created. If not, a fatal error occurs. Note that a fatal error also occurs if no composite extension record is defined.

D.4.3.2 Constraint Class Extension Structure

The extension record defined for Constraint is as follows:

```
typedef struct {
    XtPointer next_extension;
    XrmQuark record_type;
    long version;
    Cardinal record_size;
    XtArgsProc get_values_hook;
} ConstraintClassExtensionRec, *ConstraintClassExtension;
```

The record_type should be initialized to NULLQUARK. The version field should be initialized to XtConstraintExtensionVersion.

D.4.3.3 New Constraint get_values_hook method

If the Constraint widget or any of its superclasses have declared a ConstraintClassExtension record in the constraint class part extension fields with a record type of NULLQUARK and the get_values_hook field in the extension record is non-NULL, XtGetValues calls this get_values_hook method to allow the parent to return derived constraint fields. The Constraint get_values_hook method is for processing the resources of subparts.

The Constraint extension get_values_hook method is called after the Core get_values_ hook methods are called.

D.4.3.4 Shell Class Extension Structure

The extension record defined for Constraint is as follows:

```
typedef struct {
    XtPointer next_extension;
    XrmQuark record_type;
    long version;
    Cardinal record_size;
    XtGeometryHandler root_geometry_manager;
} ShellClassExtensionRec, *ShellClassExtension;
```

The symbolic constant for the ShellClassExtension version id is XtShellExtensionVersion.

The root_geometry_manager field specifies a procedure which acts as the parent geometry manager for geometry requests made by shell widgets. When a shell widget calls XtMakeGeometryRequest or XtMakeResizeRequest, the root_geometry_manager procedure is invoked to negotiate the new geometry with the window manager. If the window manager permits the new geometry, the root_geometry_manager should return XtGeometryYes; if the window manager denies the geometry request or

does not change the window geometry within some timeout interval (equal to wm_timeout in the case of WMShells), the root_geometry_manager should return XtGeometryNo. If the window manager makes some alternate geometry change, the root_geometry_manager may either return XtGeometryNo and handle the new geometry as a resize, or may return XtGeometryAlmost in anticipation that the shell will accept the compromise. If the compromise is not accepted, the new size must then be handled as a resize. Subclasses of Shell which wish to provide their own root_ geometry _manager are strongly encouraged to use enveloping to invoke their superclass' root_geometry_manager under most situations, as the window manager interaction may be very complex.

If no ShellClassPart extension record is declared with record_type equal to NULLQUARK, then XtInheritRootGeometryManager is assumed.

D.4.4 New XtNvisual Resource

Shell widgets now have a resource for the visual they will use. This is normally inherited by all their child widgets. If a visual other than the default is chosen, usually the XtNcolormap resource also must be set. See Chapter 7, *Events, Translations, and Accelerators*, in Volume One, *Xlib Programming Manual*, for a discussion of visuals and how to choose one.

D.4.5 Methods

Arguments were added to the procedure definitions for the initialize and set_values methods, to pass in the argument list specified in the function that resulted in the method invocation. The added arguments are unlikely to be used in a widget that has no subpart. The added arguments make the initialize_hook and set_values_hook functions obsolete, but they have been retained for those widgets which used them.

The Core realize method now uses CopyFromParent as the default colormap, not the default colormap of the screen.

The chaining of Composite and Constraint methods was not specified in R3, but is now. The self-contained fields in Composite widget classes are:

- geometry_manager
- change_managed
- insert_child
- delete_child
- accepts_objects

The self-contained field in Constraint widget classes is:

- constraint_size

The self-contained field in Shell widget classes are:

- `root_geometry_manager`

In addition, for subclasses of Constraint, the following fields of the `ConstraintClassPart` and `ConstraintClassExtensionRec` structures are downward chained (beginning at Constraint):

- `resources`
- `initialize`
- `set_values`
- `get_values_hook`

For subclasses of Constraint, the following field of `ConstraintClassPart` are upward chained (ending at Constraint):

- `destroy`

D.4.6 Mapping Order of Realized Child Widgets

Newly specified in R4 is the following: when calling the realize procedures for children of a composite widget, `XtRealizeWidget` calls the procedures in reverse order of appearance in the `composite.children` list. By default, this will result in the stacking order of any newly created subwindows being top-to-bottom in the order of appearance on the list and the most recently created widget will be at the bottom.

D.4.7 Callback Lists

D.4.7.1 XtNunrealizeCallback

If a widget class defines a resource `XtNunrealizeCallback` as a callback list, Xt will call any callback function registered when unrealizing a widget of that class.

D.4.7.2 Internal Format of Callback Lists

Newly documented in R4 is the fact that the internal representation of a callback list, the value of the `XtRCallback` resource, is implementation-dependent; clients may not make assumptions about the value stored in this resource. To determine whether a widget is capable of supporting a callback list, or whether callbacks have been registered, `XtHasCallbacks` can be used (not new to R4)—it returns `XtCallbackNoList`, `XtCallbackHasNone` or `XtCallbackHasSome`.

D.4.7.3 New Routine to Call Callback List

`XtCallCallbackList` executes the procedures in a callback list, specifying the callback list by address. In other words, the function calls all the procedures on the callback list without reference to the widget those callback functions are registered for. This function avoids the callback list lookup overhead.

D.4.8 Resource Lists

`XtOffsetOf` determines the byte offset of a field within a structure type. It performs the same function as `XtOffset`, but is more portable.

D.4.8.1 Base Offsets In Resource List

`XtWidgetBaseOffset` has been added as an address mode which can be used in resource lists. `XtWidgetBaseOffset` is similar to `XtBaseOffset` except that it searches for the closest windowed ancestor if the object is not a subclass of Core.

D.4.8.2 XtProcedureArg

In the field usually specified as `XtBaseOffset`, you can specify `XtProcedureArg` instead. This instructs Xt to call a procedure of type `XtConvertArgProc` to get the resource value.

D.4.8.3 Default Resource Values

The `default_address` field in each resource list element specifies the address of the default resource value. As a special case, if the `default_type` is `XtRString`, then the value in the `default_addr` field is the pointer to the string rather than a pointer to the pointer.

D.4.8.4 Getting a Constraint Resource List

`XtGetConstraintResourceList` gets the constraint resource list structure for a widget class.

D.4.9 Converters

D.4.9.1 New Xt Resource Representations

The following resource representation types are now defined by the Intrinsics.

Table D-3. New Representation Types

Resource Type	Structure or Field Type
`XtRBitmap`	`Pixmap`, depth=1
`XtRGeometry`	`char*`, format as defined by `XParseGeometry`
`XtRInitialState`	`int`
`XtRLongBoolean`	`long`
`XtRObject`	`Object`
`XtRStringTable`	`char**`
`XtRVisual`	`Visual*`

<X11/StringDefs.h> now defines the following resource types as a convenience for widgets, although they do not have any corresponding datatype assigned: `XtREditMode`, `XtRJustify` and `XtROrientation`.

D.4.9.2 New Xt Resource Converters

Xt now defines converters from `XtRString` to `XtRInitialState`, from `XtRString` to `XtRVisual`, and from `XtRString` to `XtRAtom`.

The `XtRString` to `XtRInitialState` conversion accepts the values `NormalState` or `IconicState` as defined by the Inter-Client Communications Conventions.

The `XtRString` to `XtRVisual` conversion calls `XMatchVisualInfo` using the `XtNscreen` and `XtNdepth` resources and returns the first matching `Visual` on the list. The widget resource list must be certain to specify any resource of type `XtRVisual` after the depth resource. The allowed string values are the Protocol visual class names: `StaticGray`, `StaticColor`, `TrueColor`, `GrayScale`, `PseudoColor`, and `DirectColor`.

The string to font and font structure converters no longer evaluate the constant `XtDefaultFont` to the font in the screen's default graphics context. Instead, they do the following:

1. Query the resource database for the resource whose full name is "xtDefaultFont", class "XtDefaultFont" (that is, no widget name/class prefixes) and use a type `XtRString` value returned as the font name, or a type `XtRFont` or `XtRFontStruct` value directly as the resource value.
2. If the resource database does not contain a value for xtDefaultFont, class XtDefaultFont, or if the returned font name cannot be successfully opened, an implementation-defined font in ISO8859-1 character set encoding is opened. (One possible algorithm is to

form an `XListFonts` using a wildcard font name and use the first font in the list. This wildcard font name should be as broad as possible to maximize the probability of locating a usable font; for example, "-*-*-*-R-*-*-*-120-*-*-*-*-ISO8859-1".)

3. If no suitable ISO8859-1 font can be found, issue an error message.

D.4.9.3 New Fn for Issuing Conversion Warnings

`XtDisplayStringConversionWarning` supercedes `XtStringConversionWarning` as the convenience routine for resource converters that convert from string values.

D.4.9.4 New Resource Converters Interface

A new interface declaration for resource type converters was defined to provide more information to converters, to support conversion cache cleanup with reference counting, and to allow additional procedures to be declared to free resource values. The old interfaces remain (for backwards compatibility) and a new set of procedures was defined which work only with the new type converter interface.

D.4.9.4.1 Registering a Resource Converter. To register a new converter for all application contexts in a process, use `XtSetTypeConverter`. To register a new converter in a single application context, use `XtAppSetTypeConverter`. `XtAddConverter` and `XtAppAddConverter` are superceded by these new routines. The new routines provide a `cache_type` argument that specifies when and how conversions made with this converter should be cached. Another argument specifies a procedure, called a *destructor*, that will be called to free resource values when the reference count reaches zero.

D.4.9.4.2 XtDestructor functions. To allow the Intrinsics to free cached resource values produced by type converters when widgets are destroyed or displays are closed, you may provide a resource destructor procedure. You register one in the call that registers the new-style type converter.

D.4.9.4.3 New Resource Converter Procedure Type. A resource converter procedure pointer is now of type `XtTypeConverter`. In R3 it was `XtConverter`. `XtTypeConvert` has two new arguments, *`display`*, and *`converter_data`*. The `converter_data` field specifies a location into which the converter may store converter-specific data that is associated with this conversion.

Any converter that queries the server to get the conversion should use the *`display`* argument to specify which server to query. For example, the `XtRString` to `XtRPixel` convert uses its *`display`* argument in the `XAllocColor` Xlib call it makes. The `display`

argument can also be used when generating error messages, to identify the application context (with the function XtDisplayToApplicationContext).

The new style converter now must return True if the conversion was successful and False otherwise. If the conversion cannot be performed due to an improper source value, a warning message should also be issued with XtAppWarningMsg or XtDisplayStringConversionWarning.

D.4.9.4.4 Invoking a Resource Converter. XtConvertAndStore and XtCallConverter supercede XtConvert and XtDirectConvert respectively. The new routines include caching for converted values and caching for whether the previous conversion for the given value succeeded, and better support for application contexts.

D.4.9.4.5 Caching and Reference Counting. XtAppReleaseCacheRefs explicitly decrements the reference counts for resources obtained from XtCallConverter.

As a convenience to clients needing to explicitly decrement reference counts via a callback function, the Intrinsics define two callback procedures; XtCallbackReleaseCacheRef and XtCallbackReleaseCacheRefList.

XtConvertAndStore looks up and calls a resource converter, copying the resulting value, and freeing a cached resource when a widget is destroyed. XtConvertAndStore adds XtCallbackReleaseCacheRef to the XtNdestroyCallback list of the specified widget if the conversion returns an XtCacheRef value.

By default, XtCreateWidget performs processing equivalent to XtConvertAndStore when initializing a widget instance. Because there is extra memory overhead required to implement reference counting, clients may distinguish those widgets which are never destroyed before the application exits from those which may be destroyed and whose resources should be deallocated. To specify whether or not reference counting is to be enabled for the resources of a particular widget when the widget is created, the client can specify a value for the Boolean resource XtNinitialResourcesPersistent, class XtCInitialResourcesPersistent.

When XtCreateWidget is called, XtNinitialResourcesPersistent must be set to False in either the arg list or the resource database for the resources referenced by this widget to be reference counted, regardless of how the type converter may have been registered. The effective default value is True. Only clients which expect to destroy one or more widgets and want resources deallocated should specify False for XtNinitialResourcesPersistent.

Even if XtNinitialResourcesPersistent is True, resources are still freed and destructors called when XtCloseDisplay is called if the conversion was registered as XtCacheByDisplay.

D.4.10 Keyboard Handling

D.4.10.1 KeyCode-to-KeySym Conversions

The Intrinsics maintain tables internally to map keycodes to keysyms for each open display. Translator procedures and any other client may share a single copy of this table to perform the same mapping.

`XtGetKeysymTable` returns a pointer to the keycode to keysym mapping table for a particular display.

D.4.10.2 KeySym-to-KeyCode Conversions

`XtKeysymToKeycodeList` returns the list of keycodes that map to a particular keysym in the keyboard mapping table maintained by the Intrinsics.

D.4.11 Selections

Selection requests for the TIMESTAMP target are now answered automatically by Xt.

Also, `XtGetSelectionRequest` has been added. `XtGetSelectionRequest` retrieves the `SelectionRequest` event which triggered the `convert_selection` procedure. This is necessary for some ICCCM target types. `XtGetSelectionRequest` may only be called from within a `convert_selection` procedure.

A whole set of routines for incremental transfers have been added. An incremental transfer is just like an atomic transfer, except that the functions you write to convert and get the data must expect to be called multiple times for the same selection. The basic three functions that parallel the functions provided for atomic transfers are `XtOwnSelectionIncremental`, `XtGetSelectionValueIncremental`, and `XtGetSelectionValuesIncremental`. The function types registered with these three basic functions are `XtConvertSelectionIncrProc`, `XtLoseSelectionIncrProc`, `XtSelectionDoneIncrProc`, `XtCancelConvertSelectionProc`, `XtSelectionIncrCallbackProc`, and `XtCancelSelectionCallbackProc`.

`XtLastTimestampProcessed` retrieves the timestamp from the most recent call to `XtDispatchEvent` which contained a timestamp. (`XtDispatchEvent` records the last timestamp in any event which contains a timestamp.)

D.4.12 Objects

Although widget writers are free to treat Core as the base class of the widget hierarchy, there are actually three classes above it. These classes are Object, RectObj (Rectangle Object), and the un-named class. All of these classes, their subclasses and all widgets are referred to generically as *objects*. By convention, the term *widget* refers only to a direct subclass of Core and the term *gadget* refers to a direct subclass of RectObj. A direct subclass of Object

is an object. Note that the term object is also used generically to encompass all objects, gadgets, and widgets.

The basic difference between a widget and a gadget is that a widget has a window on the server side, but a gadget doesn't. Therefore, gadgets share their parent's window. Chapter 12, *Menus, Gadgets, and Cascaded Pop Ups*, describes how gadgets work and how to use the gadgets provided with the R4 Athena Widget set. An object is also a sort of window-less widget, but it has no geometry resources, `expose` method, or sensitivity. The Object class exists to enable programmers to use the Intrinsics' classing and resource handling mechanisms for things besides widgets. Objects make obsolete many common uses of subparts and subresources.

Formal support for objects is new to Release 4. Motif 1.0 was based on a modified version of the R3 Intrinsics which included gadgets, but this version is not compatible with the R4 Consortium standard Intrinsics described here. Motif 1.1, however, is based on the R4 standard Intrinsics described here.

Composite widget classes that wish to accept gadget children must set the `accepts_objects` field in the `CompositeClassExtension` structure to `True`. `XtCreateWidget` will generate an error message otherwise.

D.4.12.1 The Gadget Parent

The Intrinsics will clear areas of a parent window obscured by RectObj children, causing `Expose` events, under these circumstances:

- A RectObj child is managed or unmanaged, and the parent is a Composite widget.
- In a call to `XtSetValues` on a RectObj child, one or more `set_values` procedures returns True in a call to `XtSetValues`, indicating that the child needs to be redisplayed.
- In a call to `XtSetValues` on a RectObj child, a change in geometry is made; areas will be cleared corresponding to both the old and the new child geometries.

When a gadget is playing the role of a widget, developers must be reminded to avoid making assumptions about the object passed in the *`Widget`* argument to a callback procedure.

The Intrinsics define an un-named class between RectObj and Core for possible future use. The content of the class record for this class and the result of an attempt to subclass or to create a widget of this un-named class are undefined.

D.4.12.2 Xt Functions that take any Object

In R3, all functions which had an argument of type `Widget` or `WidgetClass` could accept any class of widget. Now that objects have been implemented, this is no longer true. The following lists describe what classes are valid in all Xt functions that accept widget or widget class arguments or that return such values.

The `WidgetClass` arguments to the following procedures may be `objectClass` or any subclass:

`XtCheckSubclass`
`XtCreateWidget`
`XtGetConstraintResourceList`
`XtGetResourceList`
`XtInitializeWidgetClass`
`XtIsSubclass`
`XtVaCreateWidget`

The Widget arguments to the following procedures can be of class Object or any subclass:

`XtAddCallback`
`XtAddCallbacks`
`XtCallCallbacks`
`XtCheckSubclass`
`XtClass`
`XtConvert`
`XtConvertAndStore`
`XtCreateWidget`
`XtDestroyWidget`
`XtDisplayOfObject`
`XtGetApplicationResources`
`XtGetGC`
`XtGetSubresources`
`XtGetSubvalues`
`XtGetValues`
`XtHasCallbacks`
`XtIsApplicationShell`
`XtIsComposite`
`XtIsConstraint`,
`XtIsManaged` (returns `False` if not `RectObj` or subclass)
`XtIsObject`
`XtIsOverrideShell`
`XtIsRealized` (returns the state of the nearest windowed ancestor)
`XtIsRectObj`
`XtIsSensitive` (returns `False` if not RectObj or subclass)
`XtIsShell`
`XtIsSubclass`
`XtIsToplevelShell`
`XtIsTransientShell`
`XtIsVendorShell`
`XtIsWidget`
`XtIsWMShell`
`XtName`
`XtNameToWidget`
`XtReleaseGC`
`XtRemoveAllCallbacks`
`XtRemoveCallback`
`XtRemoveCallbacks`
`XtScreenOfObject`
`XtSetKeyboardFocus` (descendant)
`XtSetValues`
`XtSuperclass`
`XtVaCreateWidget`
`XtVaGetApplicationResources`
`XtVaGetSubresources`
`XtVaGetSubvalues`
`XtVaGetValues`
`XtVaSetValues`
`XtWidgetToApplicationContext`
`XtWindowOfObject`

The return value of these procedures will be of class Object or a subclass:

```
XtCreateWidget
XtVaCreateWidget
XtParent
XtNameToWidget
```

The return value of these procedures will be objectClass or a subclass: `XtClass`, `XtSuperclass`.

D.4.12.3 Xt Functions that take any RectObj

The WidgetClass arguments to the following procedures may be `rectObjClass` or any subclass: `XtCreateManagedWidget`, `XtVaCreateManagedWidget`.

The Widget arguments to the following procedures can be of class `RectObj` or any subclass:

```
XtConfigureWidget
XtMakeGeometryRequest
XtMakeResizeRequest
XtManageChild
XtManageChildren
XtMoveWidget
XtQueryGeometry
XtResizeWidget
XtSetSensitive
XtTranslateCoords
XtUnmanageChild
XtUnmanageChildren
```

The return value of the following procedures will be of class RectObj or a subclass: `XtCreateManagedWidget`, `XtVaCreateManagedWidget`.

D.4.12.4 Xt Functions that take any Core

The Widget arguments to the following procedures must be of class Core or any subclass:

```
XtAddEventHandler
XtAddGrab

XtAddRawEventHandler
XtAugmentTranslations
XtBuildEventMask
XtCallAcceptFocus
XtCallActionProc
XtCreatePopupShell
XtCreateWindow
XtDisownSelection
XtDisplay
XtGetSelectionRequest
XtGetSelectionValue
XtGetSelectionValueIncremental
XtGetSelectionValues
XtGetSelectionValuesIncremental
XtGrabButton
XtInstallAccelerators
XtInstallAllAccelerators (both
    destination and source)
XtMapWidget
XtOverrideTranslations
XtOwnSelection
XtRealizeWidget
XtRemoveEventHandler
XtRemoveGrab
XtRemoveRawEventHandler
XtResizeWindow
XtScreen
XtSetKeyboardFocus (subtree)
XtSetMappedWhenManaged
XtSetWMColormapWindows
XtUngrabButton
XtUngrabKey
XtUngrabKeyboard
```

```
XtGrabKey
XtGrabKeyboard
XtGrabPointer
XtInsertEventHandler
XtInsertRawEventHandler
XtWindow
```

```
XtUngrabPointer
XtUninstallTranslations
XtUnmapWidget
XtUnrealizeWidget
XtVaCreatePopupShell
```

The return value of the following procedure will be of class Core or a subclass: `XtWindowToWidget`.

D.4.12.5 Xt Functions that take any Composite

The Widget arguments to the following procedures must be of class Composite or any subclass: `XtCreateManagedWidget`, `XtVaCreateManagedWidget`.

D.4.12.6 Xt Functions that take any Shell or a Subclass

The `WidgetClass` arguments to the following procedures must be of class Shell or a subclass: `XtCreatePopupShell`, `XtVaCreatePopupShell`, `XtAppCreateShell`.

The Widget arguments to the following procedures must be of a subclass of Shell:

```
XtCallbackExclusiv
XtllbackNone
XCallbackNonexclusive
XtCallbackPopdown
```

```
XtPopdown
XtPopup
XtPopupSpringLoaded
```

The return value of the following procedures will be of a subclass of Shell:

```
XtAppCreateShell
XtAppInitialize
XtCreatePopupShell
XtVaCreatePopupShell
```

D.4.12.7 Macros for Getting Object Information

The display pointer, screen pointer and window of a widget or of the closest widget ancestor of a gadget are available by calling `XtDisplayOfObject`, `XtScreenOfObject`, and `XtWindowOfObject`. `XtIsObject` tells you whether a class is a subclass of Object but not of RectObj.

`XtName` returns a pointer to the instance name of the specified object. The name does not include the names of any of the object's ancestors.

D.4.13 Miscellaneous

D.4.13.1 XtNameToWidget

`XtNameToWidget` now allows an asterisk (*) to separate widget names, to match any number of intermediate widgets. The method whereby `XtNameToWidget` searches the hierarchy for a matching widget has been more carefully spelled out: see `XtNameToWidget` in Volume Five, *X Toolkit Intrinsics Reference Manual*.

D.4.13.2 New Exposure Compression Controls

The `compress_exposure` field of the Core structure can now be set to additional values to tailor the exposure compression for a particular widget. The four basic values are shown in Table D-4.

Table D-4. Flags for compress_exposure Field of Core Class Structure

Symbol	Description
`XtExposeCompressMaximal`	All expose series currently in the queue are coalesced into a single event without regard to intervening nonexposure events. If a partial series is in the end of the queue, the Intrinsics will block until the end of the series is received. This setting is not advised, since `ConfigureNotify` events may be between Expose event series.
`XtExposeCompressMultiple`	Consecutive series of exposure events are coalesced into a single event when an exposure event with count==0 is reached and either the event queue is empty or the next event is not an exposure event.
`XtExposeCompressSeries`	Each series of exposure events is coalesced into a single event when an exposure event with `count==0` is reached.
`XtExposeNoCompress`	No exposure compression is performed; every selected event is individually dispatched to the expose procedure, with a region argument of `NULL`.

Note that the first two values are different names for the settings available under R3, `False` and `True` respectively. The most useful value is `XtExposeCompressMultiple`. Any of the above values can be optionally ORed with any combination of the following flags (all implementation-defined):

Table D-5. Optional OR Flags for compress_exposure Field of Core Class Structure

Symbol	Description
XtExposeGraphicsExpose	Specifies that GraphicsExpose events are also to be dispatched to the expose method. GraphicsExpose events will be compressed, if specified, in the same manner as Expose events.
XtExposeGraphicsExposeMerged	Specifies in the case of XtExposeCompressMultiple and XtExposeCompressMaximal that series of GraphicsExpose and Expose events are to be compressed together, with the final event type determining the type of the event passed to the expose procedure. If this flag is not set, then only series of the same event type as the event at the head of the queue are coalesced. This flag also implies XtExposeGraphicsExpose.
XtExposeNoExpose	Specifies that NoExpose events are also to be dispatched to the expose procedure. NoExpose events are never coalesced with other exposure events, nor with each other.

D.4.13.3 Requesting Key and Button Grabs

Xt now provides a set of key and button grab interfaces that are parallel to those provided by Xlib. Toolkit applications and widgets that need to passively grab keys or buttons should use the following Intrinsics routines rather than the corresponding Xlib routines.

The new Xt routines are: XtGrabKey, XtUngrabKey, XtGrabKeyboard, XtUngrabKeyboard, XtGrabButton, XtUngrabButton, XtGrabPointer, and XtUngrabPointer.

Remember that these routines are interfaces to Xlib grab routines: they are not the same as the Xt grab mode which affects only the dispatching of events within your application.

To affect dispatching of events outside of your application, in R3 you needed to make a Xlib call to grab the pointer, or use MenuPopup, the standard action that made the grab for you (which is now also called XtMenuPopup). Given the new R4 grab routines, you can pop-up a menu using any of the pop-up mechanisms (XtPopup, XtPopupSpringLoaded, XtCallbackExclusive, XtCallbackNonexclusive, or XtCallbackNone)

and still get the desired grab by calling XtGrabButton, or by using XtRegisterGrabAction.

D.4.13.4 New Macros for Widget Superclass Determination

To test if a given widget is a subclass of an Intrinsics-defined class, the Intrinsics define macros or functions equivalent to XtIsSubclass (which existed in R3) for each of the built-in classes. These procedures are: XtIsObject, XtIsRectObj, XtIsWidget, XtIsComposite, XtIsConstraint, XtIsShell, XtIsOverrideShell, XtIsWMShell, XtIsVendorShell, XtIsTransientShell, XtIsTopLevelShell, and XtIsApplicationShell.

All of these macros and functions have just one argument, the object. These procedures may be faster than calling XtIsSubclass directly for the built-in classes.

D.4.13.5 Macros and Procedure Equivalents

Any Xt function may be implemented as both a macro and a procedure. Any such macro will expand to code that evaluates each of its arguments exactly once, fully protected by parentheses, so that arbitrary expressions may be used as arguments. Applications may use #undef to remove a macro definition and ensure that the actual function is referenced.

XtNew, XtNumber, XtOffsetOf, XtOffset, and XtSetArg are macros which do not have function equivalents, and may expand their arguments in a manner other than that described above. Therefore, their arguments cannot contain arbitrary expressions. Arguments for these macros should not include expressions that increment a variable, because an argument might be referenced twice.

D.4.13.6 Finding File Names

The Intrinsics provide procedures to look for a file by name using string substitutions in a list of file specifications. There are two routines provided; XtFindFile and XtResolvePathname. XtFindFile uses an arbitrary set of client-specified substitutions and XtResolvePathname uses a set of standard substitutions corresponding to the *X/Open Portability Guide* language localization conventions, and is used by Xt's language localization support.

D.4.13.7 Improved Multi-Display Support in Xmu

Multi-display programs should now be able to use the Xmu type converters. Xmu also provides utilities for managing the multiple display data structures.

E

The xbitmap Application

This appendix shows the complete code for all versions of xbitmap, *which is described in Chapters 4 and 5.*

In This Chapter:

The BitmapEdit Widget 497
The BitmapEdiP.h Private Header File 510
The BitmapEdit.h Public Header File 511
xbitmap8 513

E
The xbitmap Application

This appendix shows the complete code for the BitmapEdit widget (without selections), and the complete code for an advanced version of the *xbitmap* application (*xbitmap8* in the example distribution) which is similar to *xbitmap3* described in Chapter 4 except that it both reads and writes X11 bitmap files (*xbitmap3* was capable of writing them only).

All source code from this book is available free from numerous sources, as described in the Preface.

E.1 The BitmapEdit Widget

Example E-1. BitmapEdit: complete widget code

```
/*
 * BitmapEdit.c - bitmap editor widget.
 */

#include <X11/IntrinsicP.h>
#include <X11/StringDefs.h>

#include <stdio.h>

#include "BitmapEdiP.h"

#define INTERNAL_WIDTH     2
#define INTERNAL_HEIGHT 4

#define DEFAULT_PIXMAP_WIDTH     32     /* in cells */
#define DEFAULT_PIXMAP_HEIGHT     32     /* in cells */

#define DEFAULT_CELL_SIZE     30     /* in pixels */

/* values for instance variable is_drawn */
#define DRAWN 1
#define UNDRAWN 0

/* modes for drawing */
#define DRAW 1
#define UNDRAW 0

#define MAXLINES 1000     /* max of horiz or vertical cells */
#define SCROLLBARWIDTH 15

#define DEFAULTWIDTH 300  /* widget size when showAll is False */
```

Example E-1. BitmapEdit: complete widget code (continued)

```
#define offset(field) XtOffset(BitmapEditWidget, field)

static XtResource resources[] = {
    {
    XtNforeground,
    XtCForeground,
    XtRPixel,
    sizeof(Pixel),
    offset(bitmapEdit.foreground),
    XtRString,
    XtDefaultForeground
     },
     {
    XtNcallback,
    XtCCallback,
    XtRCallback,
    sizeof(XtPointer),
    offset(bitmapEdit.callback),
    XtRCallback,
    NULL
     },
     {
    XtNcellSizeInPixels,
    XtCCellSizeInPixels,
    XtRInt, sizeof(int),
    offset(bitmapEdit.cell_size_in_pixels),
    XtRImmediate,
    (XtPointer)DEFAULT_CELL_SIZE
     },
     {
    XtNpixmapWidthInCells,
    XtCPixmapWidthInCells,
    XtRDimension,
    sizeof(Dimension),
    offset(bitmapEdit.pixmap_width_in_cells),
    XtRImmediate,
    (XtPointer)DEFAULT_PIXMAP_WIDTH
     },
     {
    XtNpixmapHeightInCells,
    XtCPixmapHeightInCells,
    XtRDimension,
    sizeof(Dimension),
    offset(bitmapEdit.pixmap_height_in_cells),
    XtRImmediate,
    (XtPointer)DEFAULT_PIXMAP_HEIGHT
     },
     {
    XtNcurX,
    XtCCurX,
    XtRInt,
    sizeof(int),
    offset(bitmapEdit.cur_x),
    XtRImmediate,
    (XtPointer) 0
     },
```

Example E-1. BitmapEdit: complete widget code (continued)

```
    {
    XtNcurY,
    XtCCurY,
    XtRInt,
    sizeof(int),
    offset(bitmapEdit.cur_y),
    XtRString,
    (XtPointer) NULL
    },
    {
    XtNcellArray,
    XtCCellArray,
    XtRString,
    sizeof(String),
    offset(bitmapEdit.cell),
    XtRImmediate,
    (XtPointer) 0
    },
    {
    XtNshowEntireBitmap,
    XtCShowEntireBitmap,
    XtRBoolean,
    sizeof(Boolean),
    offset(bitmapEdit.showAll),
    XtRImmediate,
    (XtPointer) TRUE
    },
};

/* Declaration of methods */

static void Initialize();
static void Redisplay();
static void Destroy();
static void Resize();
static Boolean SetValues();
static XtGeometryResult QueryGeometry();

/* these Core methods not needed by BitmapEdit:
 *
 * static void ClassInitialize();
 * static void Realize();
 */

/* the following are private functions unique to BitmapEdit */
static void DrawPixmaps(), DoCell(), ChangeCellSize();

/* the following are actions of BitmapEdit */
static void DrawCell(), UndrawCell(), ToggleCell();

/* The following are public functions of BitmapEdit, declared extern
 * in the public include file: */
char *BitmapEditGetArrayString();

static char defaultTranslations[] =
    "<Btn1Down>:    DrawCell()          \n\
    <Btn2Down>:    UndrawCell()         \n\
    <Btn3Down>:    ToggleCell()         \n\
    <Btn1Motion>:  DrawCell()           \n\
```

Example E-1. BitmapEdit: complete widget code (continued)

```
    <Btn2Motion>:  UndrawCell()            \n\
    <Btn3Motion>:  ToggleCell()";

static XtActionsRec actions[] = {
        {"DrawCell", DrawCell},
        {"UndrawCell", UndrawCell},
        {"ToggleCell", ToggleCell},
};

/* definition in BitmapEdit.h */
static BitmapEditPointInfo info;

BitmapEditClassRec bitmapEditClassRec = {
    {
    /* core_class fields      */
    /* superclass             */ (WidgetClass) &coreClassRec,
    /* class_name             */ "BitmapEdit",
    /* widget_size            */ sizeof(BitmapEditRec),
    /* class_initialize       */ NULL,
    /* class_part_initialize  */ NULL,
    /* class_inited           */ FALSE,
    /* initialize             */ Initialize,
    /* initialize_hook        */ NULL,
    /* realize                */ XtInheritRealize,
    /* actions                */ actions,
    /* num_actions            */ XtNumber(actions),
    /* resources              */ resources,
    /* num_resources          */ XtNumber(resources),
    /* xrm_class              */ NULLQUARK,
    /* compress_motion        */ TRUE,
    /* compress_exposure      */ XtExposeCompressMultiple,
    /* compress_enterleave    */ TRUE,
    /* visible_interest       */ FALSE,
    /* destroy                */ Destroy,
    /* resize                 */ Resize,
    /* expose                 */ Redisplay,
    /* set_values             */ SetValues,
    /* set_values_hook        */ NULL,
    /* set_values_almost      */ XtInheritSetValuesAlmost,
    /* get_values_hook        */ NULL,
    /* accept_focus           */ NULL,
    /* version                */ XtVersion,
    /* callback_private       */ NULL,
    /* tm_table               */ defaultTranslations,
    /* query_geometry         */ QueryGeometry,
    /* display_accelerator    */ XtInheritDisplayAccelerator,
    /* extension              */ NULL
    },
    {
    /* dummy_field            */ 0,
    },
};

WidgetClass bitmapEditWidgetClass = (WidgetClass) & bitmapEditClassRec;

static void
GetDrawGC(w)
Widget w;
```

Example E-1. BitmapEdit: complete widget code (continued)

```
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground | GCDashOffset
            | GCDashList | GCLineStyle;

    /*
     * Setting foreground and background to 1 and 0 looks like a
     * kludge but isn't.  This GC is used for drawing
     * into a pixmap of depth one.  Real colors are applied with a
     * separate GC when the pixmap is copied into the window.
     */
    values.foreground = 1;
    values.background = 0;
    values.dashes = 1;
    values.dash_offset = 0;
    values.line_style = LineOnOffDash;

    cw->bitmapEdit.draw_gc = XCreateGC(XtDisplay(cw),
            cw->bitmapEdit.big_picture, mask, &values);
}

static void
GetUndrawGC(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground;

    /* this looks like a kludge but isn't.  This GC is used for drawing
     * into a pixmap of depth one.  Real colors are applied as the
     * pixmap is copied into the window.
     */
    values.foreground = 0;
    values.background = 1;

    cw->bitmapEdit.undraw_gc = XCreateGC(XtDisplay(cw),
            cw->bitmapEdit.big_picture, mask, &values);
}

static void
GetCopyGC(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    XGCValues values;
    XtGCMask mask = GCForeground | GCBackground;

    values.foreground = cw->bitmapEdit.foreground;
    values.background = cw->core.background_pixel;

    cw->bitmapEdit.copy_gc = XtGetGC(cw, mask, &values);
}

/* ARGSUSED */
static void
Initialize(treq, tnew)
Widget treq, tnew;
{
```

Example E-1. BitmapEdit: complete widget code (continued)

```
    BitmapEditWidget new = (BitmapEditWidget) tnew;
    new->bitmapEdit.cur_x = 0;
    new->bitmapEdit.cur_y = 0;

    /*
     *  Check instance values set by resources that may be invalid.
     */

    if ((new->bitmapEdit.pixmap_width_in_cells < 1) ||
        (new->bitmapEdit.pixmap_height_in_cells < 1))  {
            XtWarning("BitmapEdit: pixmapWidth and/or pixmapHeight is too
        small (using 10 x 10).");
        new->bitmapEdit.pixmap_width_in_cells = 10;
        new->bitmapEdit.pixmap_height_in_cells = 10;
    }

    if (new->bitmapEdit.cell_size_in_pixels < 5) {
        XtWarning("BitmapEdit: cellSize is too small (using 5).");
        new->bitmapEdit.cell_size_in_pixels = 5;
    }

    if ((new->bitmapEdit.cur_x < 0) ||  (new->bitmapEdit.cur_y < 0)) {
        XtWarning("BitmapEdit: cur_x and cur_y must be non-negative\
            (using 0, 0).");
        new->bitmapEdit.cur_x = 0;
        new->bitmapEdit.cur_y = 0;
    }

    if (new->bitmapEdit.cell == NULL)
        new->bitmapEdit.cell =
                XtCalloc(new->bitmapEdit.pixmap_width_in_cells
                * new->bitmapEdit.pixmap_height_in_cells, sizeof(char));

    new->bitmapEdit.pixmap_width_in_pixels =
            new->bitmapEdit.pixmap_width_in_cells *
            new->bitmapEdit.cell_size_in_pixels;

    new->bitmapEdit.pixmap_height_in_pixels =
            new->bitmapEdit.pixmap_height_in_cells *
            new->bitmapEdit.cell_size_in_pixels;

    if (new->core.width == 0) {
        if (new->bitmapEdit.showAll == False)
            new->core.width = (new->bitmapEdit.pixmap_width_in_pixels
                    > DEFAULTWIDTH) ? DEFAULTWIDTH :
                    (new->bitmapEdit.pixmap_width_in_pixels);
        else
            new->core.width = new->bitmapEdit.pixmap_width_in_pixels;
    }

    if (new->core.height == 0) {
        if (new->bitmapEdit.showAll == False)
            new->core.height = (new->bitmapEdit.pixmap_height_in_pixels
                    > DEFAULTWIDTH) ? DEFAULTWIDTH :
                    (new->bitmapEdit.pixmap_height_in_pixels);
        else
            new->core.height = new->bitmapEdit.pixmap_height_in_pixels;
    }

    CreateBigPixmap(new);
```

Example E-1. BitmapEdit: complete widget code (continued)

```
    GetDrawGC(new);
    GetUndrawGC(new);
    GetCopyGC(new);

    DrawIntoBigPixmap(new);
}

/* ARGSUSED */
static void
Redisplay(w, event)
Widget w;
XExposeEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    register int x, y;
    unsigned int width, height;
    if (!XtIsRealized(cw))
        return;

    if (event) {  /* called from btn-event or expose */
        x = event->x;
        y = event->y;
        width = event->width;
        height =  event->height;
    }
    else {        /* called because complete redraw */
        x = 0;
        y = 0;
        width = cw->bitmapEdit.pixmap_width_in_pixels;
        height = cw->bitmapEdit.pixmap_height_in_pixels;
    }

    if (DefaultDepthOfScreen(XtScreen(cw)) == 1)
        XCopyArea(XtDisplay(cw), cw->bitmapEdit.big_picture, XtWindow(cw),
            cw->bitmapEdit.copy_gc, x + cw->bitmapEdit.cur_x, y +
            cw->bitmapEdit.cur_y, width, height, x, y);
    else
        XCopyPlane(XtDisplay(cw), cw->bitmapEdit.big_picture, XtWindow(cw),
            cw->bitmapEdit.copy_gc, x + cw->bitmapEdit.cur_x, y +
            cw->bitmapEdit.cur_y, width, height, x, y, 1);

}

/* ARGSUSED */
static Boolean
SetValues(current, request, new)
Widget current, request, new;
{
    BitmapEditWidget curcw = (BitmapEditWidget) current;
    BitmapEditWidget newcw = (BitmapEditWidget) new;
    Boolean do_redisplay = False;

    if (curcw->bitmapEdit.foreground != newcw->bitmapEdit.foreground) {
        XtReleaseGC(curcw, curcw->bitmapEdit.copy_gc);
        GetCopyGC(newcw);
        do_redisplay = True;
    }

    if ((curcw->bitmapEdit.cur_x != newcw->bitmapEdit.cur_x) ||
```

Example E-1. BitmapEdit: complete widget code (continued)

```
            (curcw->bitmapEdit.cur_y != newcw->bitmapEdit.cur_y))
        do_redisplay = True;

    if (curcw->bitmapEdit.cell_size_in_pixels !=
            newcw->bitmapEdit.cell_size_in_pixels) {
        ChangeCellSize(curcw, newcw->bitmapEdit.cell_size_in_pixels);
        do_redisplay = True;
    }

    if (curcw->bitmapEdit.pixmap_width_in_cells !=
            newcw->bitmapEdit.pixmap_width_in_cells)  {
        newcw->bitmapEdit.pixmap_width_in_cells =
                curcw->bitmapEdit.pixmap_width_in_cells;
        XtWarning("BitmapEdit: pixmap_width_in_cells cannot be set\
                by XtSetValues.\n");
    }

    if (curcw->bitmapEdit.pixmap_height_in_cells !=
        newcw->bitmapEdit.pixmap_height_in_cells) {
        newcw->bitmapEdit.pixmap_height_in_cells =
               curcw->bitmapEdit.pixmap_height_in_cells;
        XtWarning("BitmapEdit: pixmap_height_in_cells cannot\
                be set by XtSetValues.\n");
    }

    return do_redisplay;
}

static void
Destroy(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    if (cw->bitmapEdit.big_picture)
        XFreePixmap(XtDisplay(cw), cw->bitmapEdit.big_picture);

    if (cw->bitmapEdit.draw_gc)
        XFreeGC(XtDisplay(cw), cw->bitmapEdit.draw_gc);

    if (cw->bitmapEdit.undraw_gc)
        XFreeGC(XtDisplay(cw), cw->bitmapEdit.undraw_gc);

    if (cw->bitmapEdit.copy_gc)
        XFreeGC(XtDisplay(cw), cw->bitmapEdit.copy_gc);

    /* NOTE!  This should only free when the application didn't
     * allocate it.  Need to add another.  */
    XtFree(cw->bitmapEdit.cell);
}

static void
DrawCell(w, event)
Widget w;
XEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    DrawPixmaps(cw->bitmapEdit.draw_gc, DRAW, cw, event);
}

static void
```

Example E-1. BitmapEdit: complete widget code (continued)

```
UndrawCell(w, event)
Widget w;
XEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    DrawPixmaps(cw->bitmapEdit.undraw_gc, UNDRAW, cw, event);
}

static void
ToggleCell(w, event)
Widget w;
XEvent *event;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    static int oldx = -1, oldy = -1;
    GC gc;
    int mode;
    int newx, newy;

    /* This is strictly correct, but doesn't
     * seem to be necessary */
    if (event->type == ButtonPress) {
        newx = (cw->bitmapEdit.cur_x + ((XButtonEvent *)event)->x) /
        cw->bitmapEdit.cell_size_in_pixels;
        newy = (cw->bitmapEdit.cur_y + ((XButtonEvent *)event)->y) /
        cw->bitmapEdit.cell_size_in_pixels;
    }
    else  {
        newx = (cw->bitmapEdit.cur_x + ((XMotionEvent *)event)->x) /
        cw->bitmapEdit.cell_size_in_pixels;
        newy = (cw->bitmapEdit.cur_y + ((XMotionEvent *)event)->y) /
        cw->bitmapEdit.cell_size_in_pixels;
    }

    if ((mode = cw->bitmapEdit.cell[newx + newy *
            cw->bitmapEdit.pixmap_width_in_cells]) == DRAWN) {
        gc = cw->bitmapEdit.undraw_gc;
        mode = UNDRAW;
    }
    else {
        gc = cw->bitmapEdit.draw_gc;
        mode = DRAW;
    }

    if (oldx != newx || oldy != newy) {
        oldx = newx;
        oldy = newy;
        DrawPixmaps(gc, mode, cw, event);
    }
}

static void
DrawPixmaps(gc, mode, w, event)
GC gc;
int mode;
Widget w;
XButtonEvent *event;
```

Example E-1. BitmapEdit: complete widget code (continued)

```
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int newx = (cw->bitmapEdit.cur_x + event->x) /
            cw->bitmapEdit.cell_size_in_pixels;
    int newy = (cw->bitmapEdit.cur_y + event->y) /
            cw->bitmapEdit.cell_size_in_pixels;
    XExposeEvent fake_event;

    /* if already done, return */
    if (cw->bitmapEdit.cell[newx + newy *
            cw->bitmapEdit.pixmap_width_in_cells] == mode)
        return;

    /* otherwise, draw or undraw */
    XFillRectangle(XtDisplay(cw), cw->bitmapEdit.big_picture, gc,
            cw->bitmapEdit.cell_size_in_pixels*newx + 2,
            cw->bitmapEdit.cell_size_in_pixels*newy + 2,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3);

    cw->bitmapEdit.cell[newx + newy * cw->bitmapEdit.pixmap_width_in_cells]
            = mode;
    info.mode = mode;
    info.newx = newx;
    info.newy = newy;

    fake_event.x = cw->bitmapEdit.cell_size_in_pixels *
            newx - cw->bitmapEdit.cur_x;
    fake_event.y = cw->bitmapEdit.cell_size_in_pixels *
            newy - cw->bitmapEdit.cur_y;
    fake_event.width = cw->bitmapEdit.cell_size_in_pixels;
    fake_event.height = cw->bitmapEdit.cell_size_in_pixels;

    Redisplay(cw, &fake_event);
    XtCallCallbacks(cw, XtNcallback, &info);
}

CreateBigPixmap(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    /* always a 1 bit deep pixmap, regardless of screen depth */
    cw->bitmapEdit.big_picture = XCreatePixmap(XtDisplay(cw),
            RootWindow(XtDisplay(cw), DefaultScreen(XtDisplay(cw))),
            cw->bitmapEdit.pixmap_width_in_pixels + 2,
            cw->bitmapEdit.pixmap_height_in_pixels + 2, 1);
}

DrawIntoBigPixmap(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int n_horiz_segments, n_vert_segments;
    XSegment segment[MAXLINES];
    register int x, y;

    XFillRectangle(XtDisplay(cw), cw->bitmapEdit.big_picture,
            cw->bitmapEdit.undraw_gc, 0, 0,
                cw->bitmapEdit.pixmap_width_in_pixels
```

Example E-1. BitmapEdit: complete widget code (continued)

```
            + 2, cw->bitmapEdit.pixmap_height_in_pixels + 2);

    n_horiz_segments = cw->bitmapEdit.pixmap_height_in_cells + 1;
    n_vert_segments = cw->bitmapEdit.pixmap_width_in_cells + 1;

    for (x = 0; x < n_horiz_segments; x++) {
        segment[x].x1 = 0;
        segment[x].x2 = cw->bitmapEdit.pixmap_width_in_pixels;
        segment[x].y1 = cw->bitmapEdit.cell_size_in_pixels * x;
        segment[x].y2 = cw->bitmapEdit.cell_size_in_pixels * x;
    }

    XDrawSegments(XtDisplay(cw), cw->bitmapEdit.big_picture,
         cw->bitmapEdit.draw_gc, segment, n_horiz_segments);

    for (y = 0; y < n_vert_segments; y++) {
        segment[y].x1 = y * cw->bitmapEdit.cell_size_in_pixels;
        segment[y].x2 = y * cw->bitmapEdit.cell_size_in_pixels;
        segment[y].y1 = 0;
        segment[y].y2 = cw->bitmapEdit.pixmap_height_in_pixels;
    }

    XDrawSegments(XtDisplay(cw), cw->bitmapEdit.big_picture,
            cw->bitmapEdit.draw_gc, segment, n_vert_segments);

    /* draw current cell array into pixmap */
    for (x = 0; x < cw->bitmapEdit.pixmap_width_in_cells; x++) {
        for (y = 0; y < cw->bitmapEdit.pixmap_height_in_cells; y++) {
            if (cw->bitmapEdit.cell[x + (y *
                    cw->bitmapEdit.pixmap_width_in_cells)] == DRAWN)
                DoCell(cw, x, y, cw->bitmapEdit.draw_gc);
            else
                DoCell(cw, x, y, cw->bitmapEdit.undraw_gc);
        }
    }
}

/* A Public function, not static */
char *
BitmapEditGetArrayString(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    return (cw->bitmapEdit.cell);
}

/* ARGSUSED */
static void
Resize(w)
Widget w;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    /* resize does nothing unless new size is bigger than entire pixmap */
    if ((cw->core.width > cw->bitmapEdit.pixmap_width_in_pixels) &&
            (cw->core.height > cw->bitmapEdit.pixmap_height_in_pixels)) {
        /*
         * Calculate the maximum cell size that will allow the
         * entire bitmap to be displayed.
         */
```

Example E-1. BitmapEdit: complete widget code (continued)

```
        Dimension w_temp_cell_size_in_pixels, h_temp_cell_size_in_pixels;
        Dimension new_cell_size_in_pixels;

        w_temp_cell_size_in_pixels = cw->core.width /
                cw->bitmapEdit.pixmap_width_in_cells;
        h_temp_cell_size_in_pixels = cw->core.height /
                cw->bitmapEdit.pixmap_height_in_cells;

        if (w_temp_cell_size_in_pixels < h_temp_cell_size_in_pixels)
            new_cell_size_in_pixels = w_temp_cell_size_in_pixels;
        else
            new_cell_size_in_pixels = h_temp_cell_size_in_pixels;

        /* if size change mandates a new pixmap, make one */
        if (new_cell_size_in_pixels != cw->bitmapEdit.cell_size_in_pixels)
            ChangeCellSize(cw, new_cell_size_in_pixels);
    }
}

static void
ChangeCellSize(w, new_cell_size)
Widget w;
int new_cell_size;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    int x, y;

    cw->bitmapEdit.cell_size_in_pixels = new_cell_size;

    /* recalculate variables based on cell size */
    cw->bitmapEdit.pixmap_width_in_pixels =
            cw->bitmapEdit.pixmap_width_in_cells *
            cw->bitmapEdit.cell_size_in_pixels;

    cw->bitmapEdit.pixmap_height_in_pixels =
            cw->bitmapEdit.pixmap_height_in_cells *
            cw->bitmapEdit.cell_size_in_pixels;

    /* destroy old and create new pixmap of correct size */
    XFreePixmap(XtDisplay(cw), cw->bitmapEdit.big_picture);
    CreateBigPixmap(cw);

    /* draw lines into new pixmap */
    DrawIntoBigPixmap(cw);

    /* draw current cell array into pixmap */
    for (x = 0; x < cw->bitmapEdit.pixmap_width_in_cells; x++) {
        for (y = 0; y < cw->bitmapEdit.pixmap_height_in_cells; y++) {
            if (cw->bitmapEdit.cell[x + (y *
                    cw->bitmapEdit.pixmap_width_in_cells)] == DRAWN)
                DoCell(cw, x, y, cw->bitmapEdit.draw_gc);
            else
                DoCell(cw, x, y, cw->bitmapEdit.undraw_gc);
        }
    }
}

static void
```

Example E-1. BitmapEdit: complete widget code (continued)

```
DoCell(w, x, y, gc)
Widget w;
int x, y;
GC gc;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
        /* otherwise, draw or undraw */
    XFillRectangle(XtDisplay(cw), cw->bitmapEdit.big_picture, gc,
            cw->bitmapEdit.cell_size_in_pixels * x + 2,
            cw->bitmapEdit.cell_size_in_pixels * y + 2,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3,
            (unsigned int)cw->bitmapEdit.cell_size_in_pixels - 3);

}

static XtGeometryResult QueryGeometry(w, proposed, answer)
Widget w;
XtWidgetGeometry *proposed, *answer;
{
    BitmapEditWidget cw = (BitmapEditWidget) w;
    answer->request_mode = CWWidth | CWHeight;

    /* initial width and height */
    if (cw->bitmapEdit.showAll == True)
        answer->width = cw->bitmapEdit.pixmap_width_in_pixels;
    else
        answer->width = (cw->bitmapEdit.pixmap_width_in_pixels >
                DEFAULTWIDTH) ? DEFAULTWIDTH :
                cw->bitmapEdit.pixmap_width_in_pixels;

    if (cw->bitmapEdit.showAll == True)
        answer->height = cw->bitmapEdit.pixmap_height_in_pixels;
    else
        answer->height = (cw->bitmapEdit.pixmap_height_in_pixels >
                DEFAULTWIDTH) ? DEFAULTWIDTH :
                cw->bitmapEdit.pixmap_height_in_pixels;

    if (  ((proposed->request_mode & (CWWidth | CWHeight))
              == (CWWidth | CWHeight)) &&
              proposed->width == answer->width &&
              proposed->height == answer->height)
        return XtGeometryYes;
    else if (answer->width == cw->core.width &&
              answer->height == cw->core.height)
        return XtGeometryNo;
    else
        return XtGeometryAlmost;
}
```

E.2 The BitmapEdiP.h Private Header File

Example E-2. BitmapEdiP.h: complete private header file

```
/*
 * BitmapEditP.h - Private definitions for BitmapEdit widget
 */

#ifndef _ORABitmapEditP_h
#define _ORABitmapEditP_h

/*
 * This include not needed unless the .c file includes IntrinsicP.h
 * after this file.   Anyway, it doesn't hurt.
 */
#include <X11/CoreP.h>

/*
 * This one is always needed!
 */
#include "BitmapEdit.h"

/* New fields for the BitmapEdit widget class record */

typedef struct {
    int make_compiler_happy;     /* keep compiler happy */
} BitmapEditClassPart;

/* Full class record declaration */
typedef struct _BitmapEditClassRec {
    CoreClassPart    core_class;
    BitmapEditClassPart    bitmapEdit_class;
} BitmapEditClassRec;

extern BitmapEditClassRec bitmapEditClassRec;

/* New fields for the BitmapEdit widget record */
typedef struct {
    /* resources */
    Pixel    foreground;
    XtCallbackList callback;/* application installed callback function(s) */
    Dimension    pixmap_width_in_cells;
    Dimension    pixmap_height_in_cells;
    int cell_size_in_pixels;
    int cur_x, cur_y;       /* position of visible corner in big pixmap */
    char *cell;             /* array for keeping track of array of bits */
    Boolean showAll;        /* whether bitmap should display entire bitmap */

    /* private state */
    Dimension    pixmap_width_in_pixels;
    Dimension    pixmap_height_in_pixels;
    Pixmap big_picture;
    GC        draw_gc;     /* one plane, for drawing into pixmap */
    GC        undraw_gc;   /* one plane, for drawing into pixmap */
    GC        copy_gc;     /* defaultdepthofscreen, for copying
                              pixmap into window */
} BitmapEditPart;

/*
 * Full instance record declaration
 */
```

Example E-2. BitmapEdiP.h: complete private header file (continued)

```
typedef struct _BitmapEditRec {
    CorePart        core;
    BitmapEditPart    bitmapEdit;
} BitmapEditRec;

#endif /* _ORABitmapEditP_h */
```

E.3 The BitmapEdit.h Public Header File

Example E-3. BitmapEdit.h: complete public header file

```
#ifndef _ORABitmapEdit_h
#define _ORABitmapEdit_h

/*
 * BitmapEdit Widget public include file
 */

/*
 * This include not needed unless the application includes
 * Intrinsic.h after this file.   Anyway, it doesn't hurt.
 */
#include <X11/Core.h>

/* Resources:
 * Name              Class         RepType     Default Value
 * ----              -----         -------     -------------
 * (from RectObj)
 * ancestorSensitive
 * x                 Position      Int         0
 * y                 Position      Int         0
 * width             Dimension     Dimension   0
 * height            Dimension     Dimension   0
 * borderWidth       BorderWidth   Int
 * sensitive         Sensitive
 *
 * (from WindowObj)
 * screen            Screen        Pointer     XtCopyScreen
 * depth             Depth         Int         XtCopyFromParent
 * colormap          Colormap      Pointer     XtCopyFromParent
 * background        Background    Pixel       White
 * backgroundPixmap  Pixmap        Pixmap      XtUnspecifiedPixmap
 * borderColor       BorderColor   Pixel       Black
 * borderPixmap      BorderPixmap  Pixmap      XtUnspecifiedPixmap
 * mappedWhenManaged MappedWhenManaged Boolean True
 * translations
 * accelerators
 *
 * (from Core)
```

Example E-3. BitmapEdit.h: complete public header file (continued)

```
 * none
 *
 * (from BitmapEdit)
 * foregroundPixel   Foreground     Pixel      Black
 * backgroundPixel   Background     Pixel      White
 * callback          Callback       Callback   NULL
 * cellSize          CellSize       Int        30
 * pixmapWidth       PixmapWidth    Int        32
 * pixmapHeight      PixmapHeight   Int        32
 */

/*
 * This public structure is used as call_data to the callback.
 * It passes the x, y position of the cell toggled (in units of
 * cells, not pixels) and a mode flag that indicates whether the
 * cell was turned on (1) or off (0).
 */
typedef struct {
    int mode;
    int newx;
    int newy;
} BitmapEditPointInfo;

#define XtNcellSizeInPixels "cellSizeInPixels"
#define XtNpixmapWidthInCells "pixmapWidthInCells"
#define XtNpixmapHeightInCells "pixmapHeightInCells"
#define XtNcurX "curX"
#define XtNcurY "curY"
#define XtNcellArray "cellArray"
#define XtNshowEntireBitmap "showEntireBitmap"

#define XtCCellSizeInPixels "CellSizeInPixels"
#define XtCPixmapWidthInCells "PixmapWidthInCells"
#define XtCPixmapHeightInCells "PixmapHeightInCells"
#define XtCCurX "CurX"
#define XtCCurY "CurY"
#define XtCCellArray "CellArray"
#define XtCShowEntireBitmap "ShowEntireBitmap"

extern char *BitmapEditGetArrayString(); /* w */
       /* Widget w; */
/* Class record constants */

extern WidgetClass bitmapEditWidgetClass;

typedef struct _BitmapEditClassRec *BitmapEditWidgetClass;
typedef struct _BitmapEditRec      *BitmapEditWidget;

#endif /* _ORABitmapEdit_h */
/* DON'T ADD STUFF AFTER THIS #endif */
```

E.4 xbitmap8

Example E-4. xbitmap8: complete application code

```
/*
 * xbitmap8.c
 */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Form.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Command.h>

#include "BitmapEdit.h"

#include <stdio.h>

#define DRAWN 1
#define UNDRAWN 0

GC draw_gc, undraw_gc, invert_gc;
Pixmap normal_bitmap;
Widget bigBitmap, showNormalBitmap, showReverseBitmap;
Dimension pixmap_width_in_cells, pixmap_height_in_cells;

static void cell_toggled();

String filename;                    /* filename to read and write */
static Boolean file_contained_good_data = False;

/* ARGSUSED */
static void
Printout(widget, client_data, call_data)
Widget widget;
XtPointer client_data, call_data; /* unused */
{
    XWriteBitmapFile(XtDisplay(widget), filename, normal_bitmap,
        pixmap_width_in_cells, pixmap_height_in_cells, 0, 0);
}

/*ARGSUSED*/
static void
Redraw_small_picture(w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
    GC gc;

    if (w == showNormalBitmap)
        gc = DefaultGCOfScreen(XtScreen(w));
    else
        gc = invert_gc;

    if (DefaultDepthOfScreen(XtScreen(w)) == 1)
        XCopyArea(XtDisplay(w), normal_bitmap, XtWindow(w),
            gc, 0, 0, pixmap_width_in_cells,
            pixmap_height_in_cells, 0, 0);
    else
```

Example E-4. xbitmap8: complete application code (continued)

```
        CopyPlane(XtDisplay(w), normal_bitmap, XtWindow(w),
            gc, 0, 0, pixmap_width_in_cells,
            pixmap_height_in_cells, 0, 0, 1);
}

String
FillCell(w)
Widget w;
{
    String cell;
    int x, y;
    XImage *image;
    cell = XtCalloc(pixmap_width_in_cells * pixmap_height_in_cells,
            sizeof(char));
    /* Convert pixmap into image, so that we can
     * read indiviual pixels */
    image = XGetImage(XtDisplay(w), normal_bitmap, 0, 0,
            pixmap_width_in_cells, pixmap_height_in_cells,
            AllPlanes, XYPixmap);

    for (x = 0; x < pixmap_width_in_cells; x++) {
        for (y = 0; y < pixmap_height_in_cells; y++) {
            cell[x + (y * pixmap_width_in_cells)] = XGetPixel(image, x, y);
        }
    }
    return(cell);
}

main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context;
    Widget topLevel, form, buttonbox, quit, output;
    Arg args[5];
    int i;
    extern exit();
    unsigned int width, height;     /* NOT Dimension: used in Xlib calls */
    int junk;
    String cell;
    static XtActionsRec window_actions[] = {
        {"redraw_small_picture", Redraw_small_picture}
    };

    String trans =
            "<Expose>:    redraw_small_picture()";

    static XrmOptionDescRec table[] =
    {
        {"-pw",             "*pixmapWidthInCells",    XrmoptionSepArg, NULL},
        {"-pixmapwidth",    "*pixmapWidthInCells",    XrmoptionSepArg, NULL},
        {"-ph",             "*pixmapHeightInCells",   XrmoptionSepArg, NULL},
        {"-pixmapheight",   "*pixmapHeightInCells",   XrmoptionSepArg, NULL},
        {"-cellsize",       "*cellSizeInPixels",      XrmoptionSepArg, NULL},
    };

    topLevel = XtVaAppInitialize(
        &app_context,       /* Application context */
```

Example E-4. xbitmap8: complete application code (continued)

```
        "XBitmap8",                 /* Application class */
        table, XtNumber(table),     /* command line option list */
        &argc, argv,                /* command line args */
        NULL,                       /* for missing app-defaults file */
        NULL);                      /* terminate varargs list */

    if (argv[1] != NULL)
        filename = argv[1];
    else {
        fprintf(stderr, "xbitmap: must specify filename on command line.\n");
        exit(1);
    }

    form = XtCreateManagedWidget("form", formWidgetClass, topLevel, NULL, 0);

    buttonbox = XtCreateManagedWidget("buttonbox", boxWidgetClass, form,
            NULL, 0);

    output = XtCreateManagedWidget("output", commandWidgetClass, buttonbox,
            NULL, 0);

    XtAddCallback(output, XtNcallback, Printout, NULL);

    quit = XtCreateManagedWidget("quit", commandWidgetClass, buttonbox,
            NULL, 0);

    XtAddCallback(quit, XtNcallback, exit, NULL);

    XtAppAddActions(app_context, window_actions, XtNumber(window_actions));

    switch (XReadBitmapFile(XtDisplay(quit),
            RootWindowOfScreen(XtScreen(quit)), filename,
            &width, &height, &normal_bitmap, &junk, &junk)) {
        case BitmapSuccess:
                file_contained_good_data = True;
                if ((pixmap_width_in_cells != width) ||
                        (pixmap_height_in_cells != height)) {
                    fprintf(stderr, "xbitmap: bitmap file dimensions\
                                do not match resource database,\
                                ignoring database.\n");
                i = 0;
                XtSetArg(args[i], XtNpixmapWidthInCells, width);   i++;
                XtSetArg(args[i], XtNpixmapHeightInCells, height);   i++;
                pixmap_width_in_cells = width;
                pixmap_height_in_cells = height;
                cell = FillCell(quit);
                XtSetArg(args[i], XtNcellArray, cell);   i++;
            }
            break;
        case BitmapOpenFailed:
            fprintf(stderr, "xbitmap: could not open bitmap file, using
                    fresh bitmap.\n");
            break;
        case BitmapFileInvalid:
            fprintf(stderr, "xbitmap: bitmap file invalid.\n");
            exit(1);
        case BitmapNoMemory:
            fprintf(stderr, "xbitmap: insufficient server memory to
                    create bitmap.\n");
            exit(1);
```

Example E-4. xbitmap8: complete application code (continued)

```
        default:
            fprintf(stderr, "xbitmap: programming error.\n");
            exit(1);
    }

    /* args are set in if and switch above if file was read */
    bigBitmap = XtCreateManagedWidget("bigBitmap",
        bitmapEditWidgetClass, form, args, i);

    XtAddCallback(bigBitmap, XtNcallback, cell_toggled, NULL);

    if (!file_contained_good_data) {
        XtVaGetValues(bigBitmap,
                XtNpixmapHeightInCells, &pixmap_height_in_cells,
                XtNpixmapWidthInCells, &pixmap_width_in_cells,
                NULL);

        normal_bitmap = XCreatePixmap(XtDisplay(quit),
                RootWindowOfScreen(XtScreen(quit)),
                pixmap_width_in_cells, pixmap_height_in_cells, 1);
    }

    set_up_things(topLevel);

    showNormalBitmap = XtVaCreateManagedWidget("showNormalBitmap",
            widgetClass, buttonbox,
            XtNwidth, pixmap_width_in_cells,
            XtNheight, pixmap_height_in_cells,
            XtNtranslations, XtParseTranslationTable(trans),
            NULL);

    showReverseBitmap = XtVaCreateManagedWidget("showReverseBitmap",
            widgetClass, buttonbox,
            XtNwidth, pixmap_width_in_cells,
            XtNheight, pixmap_height_in_cells,
            XtNtranslations, XtParseTranslationTable(trans),
            NULL);

    XtRealizeWidget(topLevel);
    XtAppMainLoop(app_context);
}

set_up_things(w)
Widget w;
{
    XGCValues values;

    values.foreground = 1;
    values.background = 0;

    /* note that normal_bitmap is used as the drawable because it
     * is one bit deep.  The root window may not be one bit deep. */
    draw_gc = XCreateGC(XtDisplay(w), normal_bitmap,
            GCForeground | GCBackground, &values);

    values.foreground = 0;
    values.background = 1;
    undraw_gc = XCreateGC(XtDisplay(w), normal_bitmap,
            GCForeground | GCBackground, &values);

    /* this GC is for copying from the bitmap
```

Example E-4. xbitmap8: complete application code (continued)

```
     * to the small reverse widget */
    values.foreground = WhitePixelOfScreen(XtScreen(w));
    values.background = BlackPixelOfScreen(XtScreen(w));
    invert_gc = XtGetGC(w, GCForeground | GCBackground, &values);
}

/* ARGSUSED */
static void
cell_toggled(w, client_data, info)
Widget w;
XtPointer client_data;     /* unused */
XtPointer info;            /* call_data (from widget) */
{
    BitmapEditPointInfo *cur_info = (BitmapEditPointInfo *) info;
    /*
     * Note: BitmapEditPointInfo is defined in BitmapEdit.h
     */

    XDrawPoint(XtDisplay(w), normal_bitmap, ((cur_info->mode == DRAWN)
            ? draw_gc : undraw_gc), cur_info->newx, cur_info->newy);

    Redraw_small_picture(showNormalBitmap);
    Redraw_small_picture(showReverseBitmap);
}
```

F

Sources of Additional Information

This appendix describes where you can get more information about the X Toolkit and about X in general, including other books on the subject and the various ways to get the source code for X.

In This Chapter:

Getting the X Software 521
  Bug Fixes 522
    Notes 524
    Fairness 524
Netnews 525
Training and Consulting 525
  Phone Support 526
The X Consortium 526
Finding Out for Yourself 527

F
Sources of Additional Information

This appendix lists a few of the official and unofficial sources for information about the X Window System and associated software.

Note that some of this detailed information may become dated rather quickly. The best source of current information is the *comp.windows.x* network news group, described later in this appendix.

This book documents Release 4, the current public release level of the X software as of June 1990. Even so, many people continue to use R3 since there is a considerable delay between the date that MIT distributes a new release and the date by which vendors integrate that release into their own products and issue updates. An R3 version of this manual is available while demand is sufficient to continue reprinting. (All changes to Xt in R4 are backwards compatible so that any software that runs under R3 will also run under R4. But the converse is not true.)

F.1 Getting the X Software

You can get the X software directly from MIT on three 9-track 1600 BPI magtapes written in UNIX *tar* format or on one 9-track 6250 BPI magtape, along with printed copies of MIT's manuals, by sending a check in U.S. currency for U.S. $400 to:

MIT Software Distribution Center
Technology Licensing Office
MIT E32-300
77 Massachusetts Avenue
Cambridge, MA 02139

Their telephone number is (617) 253-6966, and the "X Ordering Hotline" is (617) 258-8330. If you want the tapes and manuals shipped overseas, the price is $500. The manual set alone is $125 including U.S. shipping or $175 including overseas shipping.

Other distribution media or formats are not available from the MIT Software Distribution Center, but are from other independent vendors such as ICS, mentioned later. The Release tape comes with source code for sample servers for Sun, HP, IBM, Apollo, Sony, DEC and several other workstations, source code for clients written by MIT, sources for the toolkits

Xt, XView (in R4), Interviews, and Andrew, contributed software written outside MIT (including the examples in this book), and sources and postscript files for all MIT's documentation. Note that the servers supplied are sample servers only; commercial vendors typically release optimized (faster) servers for the same machines.

Sites that have access to the Internet can retrieve the distribution from the following machines using anonymous *ftp*. Here are the current sites:

Location	Hostname	Address	Directory
Western USA	*gatekeeper.dec.com*	16.1.0.2	*pub/X11/R4*
Eastern USA	*uunet.uu.net*	192.48.96.2	*X/R4*
Northeastern USA	*expo.lcs.mit.edu*	18.30.0.212	*pub/R4*
	crl.dec.com	192.58.206.2	*pub/X11/R4*
Central USA	*mordred.cs.purdue.edu*	128.10.2.2	*pub/X11/R4*
	giza.cis.ohio-state.edu	128.146.8.61	*pub/X.V11R4*
Southern USA	*wuarchive.wustl.edu*	129.252.135.4	*packages/X.V11R4*
UK (Janet)	*src.doc.ic.ac.uk*	129.31.81.36	*X.V11R4*
Australia	*munnari.oz.au*	128.250.1.21	*X.V11/R4*

DO NOT do anonymous *ftp* during normal business hours, and please use the machine nearest you.

The distribution is also available by UUCP from UUNET, for sites without Internet access. The files are split up to be small enough for UUCP distribution. See the preface for instructions on getting files from UUNET if you are not a UUNET subscriber.

F.1.1 Bug Fixes

Critical bug fixes as well as a limited number of important new features are available from the archive server *xstuff@expo.lcs.mit.edu*. Electronic mail sent to this address is forwarded to a program which responds with the requested information. The rest of this section and the two that follow it (entitled Notes and Fairness) explain how to use *xstuff*.

The *xstuff* server is a mail-response program. This means that you mail it a request, and it mails back the response.

The *xstuff* server is a very dumb program. It does not have much error checking. If you don't send it commands that it understands, it will just answer "I don't understand you."

The *xstuff* server reads your entire message before it does anything, so you can have several different commands in a single message. It treats the "Subject:" header line just like any other line of the message. You can use any combination of upper and lower case letters in the commands.

The archives are organized into a series of directories and subdirectories. Each directory has an index, and each subdirectory has an index. The top-level index gives you an overview of what is in the subdirectories, and the index for each subdirectory tells you what it contains.

If you are bored with reading documentation and just want to try something, then send the server a message containing the line:

```
send index fixes
```

When you get the index back, it will contain the numbers of all of the fixes and batches of fixes in the archive. Then you can send the server another message asking it to send you the fixes that you want:

```
send fixes 1 5 9 11-20
```

If you are using a mailer that understands "@" notation, send to *xstuff@expo.lcs.mit.edu*. If your mailer deals in "!" notation, try sending to *{someplace}!mit-eddie!expo.lcs.mit.edu!xstuff*. For other mailers, you're on your own.

The server has four commands. Each command must be the first word on a line.

help The command *help* or *send help* causes the server to send you the help file. No other commands are honored in a message that asks for help (the server figures that you had better read the help message before you do anything else).

index If your message contains a line whose first word is *index*, then the server will send you the top-level index of the contents of the archive. If there are other words on that line that match the name of subdirectories, then the indexes for those subdirectories are sent instead of the top-level index. For example, you can say:

```
index
```

or:

```
index fixes
```

You can then send back another message to the *xstuff* server, using a *send* command (see below) to ask it to send you the files whose name you learned from that list.

index fixes and *send index fixes* mean the same thing: you can use *send* instead of *index* for getting an index.

If your message has an *index* or a *send index* command, then all other *send* commands will be ignored. This means that you cannot get an index and data in the same request. This is so that index requests can be given high priority.

send If your message contains a line whose first word is *send*, then the *xstuff* server will send you the item(s) named on the rest of the line. To name an item, you give its directory and its name. For example:

```
send fixes 1-10
```

Once you have named a category, you can put as many names as you like on the rest of the line. They will all be taken from that category. For example:

```
send fixes 1-10 11-20 21-30
```

Each *send* command can reference only one directory. If you would like to get one fix and one of something else, you must use two *send* commands.

You may put as many *send* commands as you like into one message to the server, but the more you ask for, the longer it will take to receive. See the Fairness Section below, for an explanation. Actually, it's not strictly true that you can put as many *send* commands as you want into one message. If the server must use UUCP mail to send your files, then it cannot send more than 100K bytes in one message. If you ask for more than it can send, then it will send as much as it can and ignore the rest.

path The *path* command exists to help in case you do not get responses from the server when you mail to it.

Sometimes the server is unable to return mail over the incoming path. There are dozens of reasons why this might happen, and if you are a true wizard, you already know what those reasons are. If you are an apprentice wizard, you might not know all the reasons, but you might know a way to circumvent them.

If you put in a *path* command, then everything that the server mails to you will be mailed to that address, rather than to the return address on your mail. The server host *expo.lcs.mit.edu* does not have a direct UUCP connection to anywhere; you must go through *mit-eddie* (the UUCP name of *eddie.mit.edu*) or somewhere else.

F.1.1.1 Notes

The *xstuff* server acknowledges every request by return mail. If you don't get a message back in a day or two you should assume that something is going wrong, and perhaps try a *path* command.

The *xstuff* server does not respond to requests from users named *root*, *system*, *daemon*, or *mailer*. This is to prevent mail loops. If your name is "Bruce Root" or "Jane Daemon", and you can document this, I will happily rewrite the server to remove this restriction. Yes, I know about Norman Mailer and Waverley Root. Norman doesn't use netmail and Waverley is dead.

F.1.1.2 Fairness

The *xstuff* server contains many safeguards to ensure that it is not monopolized by people asking for large amounts of data. The mailer is set up so that it will send no more than a fixed amount of data each day. If the work queue contains more requests than the day's quota, then the unsent files will not be processed until the next day. Whenever the mailer is run to send its day's quota, it sends the requests out shortest-first.

If you have a request waiting in the work queue and you send in another request, the new request is added to the old one (thereby increasing its size) rather than being filed anew. This prevents you from being able to send in a large number of small requests as a way of beating the system.

The reason for all of these quotas and limitations is that the delivery resources are finite, and there are many people who would like to make use of the archive.

F.2 Netnews

The Usenet network newsgroups and mailing lists are probably the most valuable source of information about X. The current list of public mailing lists that discuss X is as follows:

News Group	Description
motif@alphalpha.com	People interested in the OSF's Motif X toolkit This mailing list is also gatewayed to the Usenet newsgroup *comp.windows.x.motif*. If you receive that newsgroup, you don't need to get this mailing list.
xl1-3D@expo.lcs.mit.edu	People interested in X and 3-D graphics
x-ada@expo.lcs.mit.edu	X and ada
ximage@expo.lcs.mit.edu	People interested in image processing and X
xpert@expo.lcs.mit.edu	General discussion of X This mailing list is also gatewayed to the Usenet newsgroup *comp.windows.x*. If you receive that newsgroup, you don't need to get this mailing list.
xvideo@expo.lcs.mit.edu	Discussion of video extensions for X

The developers of X post notices of fixes to the software to the newsgroup *comp.window.x*, and this is where users and developers around the world ask and answer questions.

F.3 Training and Consulting

Numerous independent vendors provide courses on X programming. Several sources that we are aware of include:

- *Integrated Computer Solutions* (ICS), 163 Harvard Street, Cambridge, MA 02139; (617) 547-0510. Courses on Xlib, Motif, strategic overviews of X. Also provides consulting services and manages an X user's group.
- *Hands-On Learning*, 27 Cambridge Street, Burlington, MA 01803; (617) 272-0088. Courses on Xlib and Xt.
- X tutorials are now a regular feature of UNIX conventions such as the UNIX EXPO, Usenix, Uniforum, the annual X conference at MIT, and Xhibition, the conference organized by ICS. Also contact commercial server vendors for information on courses they offer.

Certain staff members at O'Reilly and Associates have developed training materials and can give training lectures or consult when time permits.

Training companies wishing to be listed here should send us information on the courses they offer.

F.3.1 Phone Support

X programming is a very new field, and since everyone who knows enough to help you is overworked, you are likely to hear "you're on your own" if you try to call for help. There are no support lines from the developers of X, because X was developed by a university, not a system manufacturer or software house. When X becomes widely supported as a commercial product, the situation should change as vendors offer support.

ICS provides phone support for a fee. See Section G.3 for their telephone number.

F.4 The X Consortium

The X Consortium can be reached at:

MIT X Consortium
545 Technology Square Rm. 217
Cambridge, MA 02139

The consortium's phone number is (617) 253-8861; its current members are shown below.

ACER Counterpoint, Inc.
AT&T
Adobe Systems
Advanced Graphics Engineering
Apollo Computer, Inc.
Apple Computer, Inc.
Ardent Computer
BULL MTS
CETIA
CalComp
Canterbury University, England
Carnegie Mellon University
Codonics, Inc.
Control Data Corporation
Cray Research, Inc.
Data General
Digital Equipment Corp.
Eastman Kodak Company
Evans & Sutherland
Fujitsu America, Inc.
GfxBase
Hewlett Packard Company
IBM Corporation
INESC*
Integrated Computer Solutions, Inc.
Integrated Solutions, Inc.
Matrox International
Megatek Corp.
Mitsubishi Electric Corporation
NCR Corporation
NEC Corporation
NTT Corporation
Network Computing Devices, Inc.
Nova Graphics International
O'Reilly & Associates, Inc.
OMRON Tateisi Electronics
Open Software Foundation
PCS Computer Systeme GmbH
Prime Computer, Inc.
Reuters
Samsung Software America
Sequent Computer Systems Inc.
Siemens AG
Silicon Graphics Computer Systems
Societe de Gestion et d'Informatique Publicis
Software Productivity Consortium
Solbourne Computer Inc.
Sony Corporation
Stanford University
Stellar Computer Inc.
Sun Microsystems, Inc.
Tatung Science and Technology

*Instituto de Engenharia de Sistemas e Computadores

Interactive Development Environments	Tektronix, Inc.
Interactive Systems Corp.	Texas Instruments
Jupiter Systems	UNICAD, Inc.
Key Systems Engineering Corp.	Unisys Corp.
Landmark Graphics Corp.	University of Lowell
Locus Computing Corp.	Visual Technology, Inc.
MIPS Computer Systems	Wang Laboratories
MITRE Corp.	X/Open Company Ltd.

Most of these companies are preparing products based on X. It should not be long before many different products are available that support X.

F.5 Finding Out for Yourself

If, heaven forbid, you don't find an answer to your problem in one of the books in our X Window System Series, and you can't get anyone to help you, what do you try next?

X is unusual in that the source code is freely available. It should be possible for most X programmers to get a copy of the X source code from the sources listed above. There you can look at some examples of how others have used X to write applications. In "Star Wars," the saying was "Use the Force, Luke." In X, it is "Use the Source, Luke." However, you may find that many of the applications on the release tape use out-of-date techniques, because they were written *before* the release.

Whenever looking at the code for the server or any of the programming libraries such as Xlib or Xt, however, remember that the X standards are defined by documents, not by code. In other words, don't depend on any of the undocumented details of the code you happen to have access to, since some other vendor may not implement the standard in the same way.

Nevertheless, it can sometimes be useful to see how things are actually being done. Once you understand how the code is organized, you can look up certain details about how X works as long as you have a good knowledge of C and a little persistence.

Xlib and the server are two distinct chunks of code. Each contains code for sending and receiving information to and from the other over the network using protocol requests, replies, events, and errors. The source tree as supplied on the X distribution tape places the Xlib source in the directory *base/lib/X*, where *base* is the top of the entire source tree. Their server source is placed in *base/server*.

The procedure for finding out something about an Xlib routine is normally to search for the routine in the Xlib code, and then figure out what it does. Sometimes the answer can be found there. Many of the routines, however, simply place their arguments in a protocol request and send it to the server. Then you will have to look in the server code for the answer. To find the correct place in the server code, you will need the symbol for the protocol request, which is the first argument in the `GetReq` call.

The server code is much more involved than Xlib itself. The device-dependent portions are in *base/server/ddx* and the device-independent portions are in *base/server/dix*. The device-independent code should be your first stop, because it is here that protocol requests from Xlib

arrive and are dispatched to the appropriate code. Search for the protocol request symbol you found in Xlib. It will appear in several source files. Start with the occurrence in *dispatch.c*, and try to figure out what the code does. This will require following leads to other routines.

If you don't find a routine in *base/server/dix*, then it must be in the device-dependent code. *base/server/ddx* has one directory in it for each brand of hardware to which a sample server has been ported. It also contains the directories */mi*, */cfb*, */mfb*, and */snf*, which contain routines used in writing the sample server device-dependent code. Note that servers may include code ostensibly for other machines.

Glossary

X uses many common terms in unique ways. A good example is "children." While most, if not all, of these terms are defined where they are first used in this manual, you will undoubtedly find it easier to refresh your memory by looking for them here.

Glossary

This glossary is an expanded version of the glossary from Volume One, *Xlib Programming Manual* (which in turn is based on the glossary in the *Xlib–C Language X Interface*, by Jim Gettys, Ron Newman, and Bob Scheifler). As such, it contains definitions of many Xlib terms not actually used in this book, but which you might come across in other reading, or in comments in code. In some cases, these Xlib terms may be used in the definitions of the Xt terms given in this glossary. Any term used in a definition, for which another entry exists in the glossary, is generally shown in italics.

accelerator

An *accelerator* is a *translation* that maps events in one widget to actions in another. The name is based on the most frequent use for this feature, namely to provide keyboard shortcuts to invoke application or widget functions that would otherwise have only a pointer-driven interface.

accept_focus method

The *accept_focus method* of a child is invoked when a parent offers the *keyboard focus* to a child by calling `XtCallAcceptFocus`. This method is part of the Core widget class.

access control list

X maintains lists of hosts that are allowed access to each server controlling a display. By default, only the local host may use the display, plus any hosts specified in the *access control list* for that display. This access control list can be changed by clients on the local host. Some server implementations may implement other authorization mechanisms in addition to or instead of this one. The list can currently be found in */etc/X#.hosts* where # is the number of the display, usually 0 (zero). The access control list is also known as the host access list.

action An *action* is a function bound by a *translation*, to be invoked in response to a user event.

actions table

An *actions table* is an array of function pointers and corresponding strings by which *actions* can be referenced in a *translation table*. The use of actions requires a widget to define both an actions table and a translation table.

active grab

A grab is *active* when the pointer or keyboard is actually owned by a single grabbing client. See also *grab*.

ancestor

If window *W* is an *inferior* of window *A*, then *A* is an *ancestor* of *W*. The *parent* window, the parent's parent window, and so on are all ancestors of the given window. The *root window* is the ancestor of all windows on a given screen.

application context

An *application context* specifies a connection to a server. When an application program has connections to multiple servers, the application context coordinates events and their dispatching, so all connections get processed.

argument list

An *argument list* is used in a call to create a widget in order to "hardcode" the value of widget resources, and also in calls to `XtSetValues` or `XtGetValues`. It consists of an array of `Arg` structures, each consisting of a resource name and the value to which it should be set.

Athena widget

MIT distributes a set of widgets developed by MIT's Project Athena in the *Athena Widget* library, Xaw. The include files for Athena widgets reside in */usr/include/X11* under Release 3 and *usr/include/X11/Xaw* under Release 4.

atom An *atom* is a unique numeric ID corresponding to a string name. Atoms are used to identify properties, types, and selections in order to avoid the overhead of passing arbitrary-length strings over the network. See also *property*.

background

Windows may have a *background*, consisting of either a solid color or a tile pattern. If a window has a background, it will be repainted automatically by the server whenever there is an `Expose` event on the window. If a window does not have a background, it will be transparent. By default, a window has a background. See also *foreground*.

backing store

When a server maintains the contents of a window, the off-screen saved pixels are known as a *backing store*. This feature is not available on all servers. Even when available, the server will not maintain a backing store, unless told to do so with a window attribute. Use the `DoesBackingStores` Xlib macro to determine if this feature is supported.

bit gravity

When a window is resized, the contents of the window are not necessarily discarded. It is possible to request the server (though no guarantees are made) to relocate the previous contents to some region of the resized window. This attraction of window contents for some location of a window is known as *bit gravity*. For example, an application that draws a graph might request that the contents be moved into the lower-left corner, so that the origin of the graph will still appear in the lower-left corner. See also *window gravity*.

bit plane

On a color or gray scale display, each pixel has more than one bit defined. Data in display memory can be thought of either as pixels (multiple bits per pixel) or as bit planes (one bit plane for each usable bit in the pixel). The *bit plane* is an array of bits the size of the screen.

bitmap A *bitmap* is a pixmap with a depth of one bit. There is no bitmap type in X11. Instead, use a pixmap of depth 1. See also *pixmap*.

border A window can have a border that is zero or more pixels wide. If a window has a border, the border can have a solid color or a tile pattern, and it will be repainted automatically by the server whenever its color or pattern is changed or an `Expose` event occurs on the window.

button grabbing

A pointer grab that becomes active only when a specified set of keys and/or buttons are held down in a particular window is referred to as a *button grab*.

byte order

The order in which bytes of data are stored in memory is hardware-dependent. For pixmaps and bitmaps, *byte order* is defined by the server, and clients with different native byte ordering must swap bytes as necessary. For all other parts of the protocol, the byte order is defined by the client, and the server swaps bytes as necessary.

callback A *callback* is an application function registered with a widget by the application using either of the calls `XtAddCallback` or `XtAddCallbacks` or through an argument list. A widget declares one or more *callback lists* as resources; applications add functions to these lists in order to link widgets to applications code.

change_managed method

When a parent should change its managed widgets, the *change_managed method* is invoked, at which time a parent reorganizes its children. The `change_managed` method is part of the Composite widget class.

child, children

1) A widget created by `XtCreateWidget` is a *child* of the widget specified as its *parent*. The parent controls the layout of its children.

2) The *children* of a window are its first-level subwindows. All of these windows were created with the same window as parent. A client creates its top-level window as a child of the root window.

class 1) (X Toolkit) A widget's *class* determines what methods will be called for it and what instance variables it has. For widget users, a widget's class is declared in the *.h* file for the widget.

2) (Xlib) There are two uses of the term *class* in X: window class and visual class. The window class specifies whether a window is `InputOnly` or `InputOutput`. The visual class specifies the color model that is used by a window. See the classes `DirectColor`, `GrayScale`, `PseudoColor`, `StaticColor`, `StaticGray`, and `TrueColor`. Both window class and visual class are set permanently when a window is created.

class_initialize method

This method—part of the Core widget—is invoked when a widget class is initialized. That is, it is called when the first instance of a particular class is created.

client An application program connects to the window system server by an interprocess communication (IPC) path, such as a TCP connection or a shared memory buffer. This program is referred to as a *client* of the window system server. More precisely, the client is the IPC path itself; a program with multiple paths open to one or more servers is viewed by the protocol as multiple clients. X Resources are available only as long as the connection remains intact, not as long as a program remains running. Normally the connection and the program terminate concurrently, but the client's resources may live on if `XChangeCloseDownMode` has been called.

clipping region

In many graphics routines, a bitmap or list of rectangles can be specified to restrict output to a particular region of the window. The image defined by the bitmap or rectangles is called a *clipping region*, or clip mask. Output to child windows is automatically clipped to the borders of the parent unless `subwindow_mode` of the GC is `IncludeInferiors`. Therefore the borders of the parent can be thought of as a clipping region.

colorcell An entry in a colormap is known as a *colorcell.* An entry contains three values specifying red, green, and blue intensities. These values are always 16-bit unsigned numbers, with zero being minimum intensity. The values are truncated or scaled by the server to match the display hardware. See also *colormap*.

colormap

A *colormap* consists of a set of colorcells. A pixel value indexes into the colormap to produce intensities of Red, Green, and Blue to be displayed. Depending on hardware limitations, one or more colormaps may be installed at one time, such that windows associated with those maps display with true colors. Regardless of the number of installable colormaps, any number of virtual colormaps can be created. When needed, a virtual colormap can be installed and the existing installed colormap might have to be uninstalled. The colormap on most systems is a limited resource that should be conserved by allocating read-only colorcells whenever possible, and by selecting RGB values from the predefined color database. Read-only cells may be shared between clients. See also *colorcell*, `DirectColor`, `GrayScale`, `PseudoColor`, `StaticColor`, `StaticGray`, and `TrueColor`.

Composite widget
A *Composite widget* is designed to manage the geometry of children; that is, a Composite widget instance can be passed in the parent argument to `XtCreateWidget`.

connection
The communication path between the server and the client is known as a *connection*. A client usually (but not necessarily) has only one connection to the server over which requests and events are sent.

Constraint widget
The *Constraint widget* is a subclass of Composite. A Constraint widget has more information about each child than a Composite widget.

constraints
A Constraint widget provides a list of resources, or *constraints*, for its children. The constraints give the Constraint widget information about how each child should be layed out.

containment
A window *contains* the pointer if the window is viewable and if the hotspot of the cursor is within a visible region of the window or a visible region of one of its inferiors. The border of the window is included as part of the window for containment. The pointer is in a window if the window contains the pointer but no inferior contains the pointer.

coordinate system
The *coordinate system* has x horizontal and y vertical, with the origin (0, 0) at the upper-left. Coordinates are discrete, and in terms of pixels. Each window and pixmap has its own coordinate system. For a window, the origin is inside the border, if there is one. The position of a child window is measured from the origin of the parent to the outside corner of the child (not the child's origin).

Core widget
The *Core widget* is the basic class in the Toolkit. All widgets that can be displayed are subclasses of Core.

cursor A *cursor* is the visible shape of the pointer on a screen. It consists of a hotspot, a shape bitmap, a mask bitmap, and a pair of pixel values. The cursor defined for a window controls the visible appearance of the pointer when the pointer is in that window.

delete_child method
The *delete_child method* is invoked on a parent after its child is deleted. This method is part of the Composite widget and is usually inherited.

depth The *depth* of a window or pixmap is the number of bits per pixel.

dereference
To access the contents of a pointer, you must *dereference* it.

descendants
See *inferiors*.

destroy method
The *destroy method* is invoked when a widget has been destroyed. This method is part of the Core widget and is used to deallocate memory and GCs.

device Keyboards, mice, tablets, track-balls, button boxes, etc. are all collectively known as input *devices*.

DirectColor
DirectColor is a visual class in which a pixel value is decomposed into three separate subfields for colormap indexing. One subfield indexes an array to produce red intensity values; the second subfield indexes a second array to produce blue intensity values; and the third subfield indexes a third array to produce green intensity values. The RGB (red, green, and blue) values in the colormap entry can be changed dynamically. This visual class is normally found on high-performance color workstations.

display A *display* is a set of one or more *screens* that are driven by a single X server. The Xlib `Display` structure contains all information about the particular display and its screens as well as the state that Xlib needs to communicate with the display over a particular connection. In Xlib, a pointer to a `Display` Structure is returned by `XOpenDisplay`. In most Xt applications, the `Display` is part of the *application context*, and need not be referenced directly. If necessary, a display can be opened directly with a call to `XtOpenDisplay`, and a pointer to the currently open display can be returned by the `XtDisplay` macro.

drawable
Both windows and pixmaps may be used as destinations in graphics operations. These are collectively known as *drawables*.

encapsulation
Encapsulation is a key concept in object-oriented programming. Objects are defined in such a way that their internals are hidden from the programs that use them; the only way to access an object should be through its published interfaces.

event An X *event* is a data structure sent by the server that describes something that just happened that may be of interest to the application. There are two major categories of events: user input and window system side effects. For example, the user pressing a keyboard key or clicking a mouse button generates an event; a window being moved on the screen also generates events—possibly in other applications as well if the movement changes the visible portions of their windows. It is the server's job to distribute events to the various windows on the screen.

event compression
Event compression is an Xt feature that allows some events to be ignored or repackaged before they are given to a widget. This happens on the client side, rather than in the server.

event handler

An *event handler* is a function that is called by Xt when a particular event is received. Event handlers have the same purpose as translations and actions—to call a function in response to an event—but event handlers have lower overhead and are not user-configurable.

event mask

Events are requested relative to a window or widget. The set of event types a client requests relative to a window is described using an *event mask*, a bitwise OR of defined symbols specifying which events are desired.

The `event_mask` is a window attribute, which can be set in Xlib with `XSelectInput`, and is also specified in calls that grab the pointer or keyboard. The `do_not_propagate_mask` attribute is also an event mask, and it specifies which events should not be propagated to ancestor windows. In Xt, you never need to set a window's `event_mask` or `do_not_propogate_mask` directly. The translation manager automatically selects the required events.

event propagation

Device-related events *propagate* from the source window to ancestor windows until a window that has selected that type of event is reached, or until the event is discarded explicitly in a `do_not_propagate_mask` attribute.

event source

The smallest window containing the pointer is the *source* of a device-related event.

expose method

When `Expose` events are received, the Intrinsics invoke the *expose method*. A widget should perform its redisplay activities in this method. The `expose` method is part of the Core widget class.

exposure

Window *exposure* occurs when a window is first mapped, or when another window that obscures it is unmapped, resized, or moved. Servers do not guarantee to preserve the contents of windows when windows are obscured or reconfigured. `Expose` events are sent to clients to inform them when contents of regions of windows have been lost and need to be regenerated.

extension

Named *extensions* to the core protocol can be defined to extend the system. Extension to output requests, resources, and event types are all possible, and expected. Extensions can perform at the same level as the core Xlib.

font A *font* is an array of characters or other bitmap shapes such as cursors. The protocol does no translation or interpretation of character sets. The client simply indicates values used to index the font array. A font contains additional metric information to determine intercharacter and interline spacing.

foreground
The pixel value that will actually be used for drawing pictures or text is referred to as the *foreground*. The foreground is specified as a member of a graphics context. See also *background*.

frozen events
Clients can *freeze* event processing while they change the screen, by grabbing the keyboard or pointer with a certain mode. These events are queued in the server (not in Xlib) until an `XAllowEvents` call with a counteracting mode is given.

GC The term *GC* is used as a shorthand for graphics context. See *graphics context*.

geometry management
A Composite parent controls its children through *geometry management*, thereby manipulating the size and placement of widgets within a window on the display.

get_values_hook method
The *get_values_hook method* is invoked by the Intrinsics when someone calls `XtGetValues`. This method is part of the Core widget class. Its role is to get the resource values of a subpart using `XtGetSubvalues`.

glyph A *glyph* is an image, usually of a character in a font, but also possibly of a cursor shape or some other shape.

grab Keyboard keys, the keyboard, pointer buttons, the pointer, and the server can be *grabbed* for exclusive use by a client, usually for a short time period. In general, these facilities help implement various styles of user interfaces. Pop-up widgets generally make a *passive grab*. Server grabs are generally used only by window managers.

graphics context
Various information for interpreting graphics primitives is stored in a *graphics context* (GC), such as foreground pixel, background pixel, line width, clipping region, etc. Everything drawn to a window or pixmap is modified by the GC used in the drawing request.

graphics primitive
A *graphics primitive* is an Xlib call that sends a protocol request to the server, instructing the server to draw a particular shape at a particular position. The graphics context specified with the primitive specifies how the server interprets the primitive.

gravity See *bit gravity* and *window gravity*.

GrayScale
GrayScale is a visual class in which the red, green, and blue values in any given colormap entry are equal, thus producing shades of gray. The gray values can be changed dynamically. `GrayScale` can be viewed as a degenerate case of `PseudoColor`.

hint Certain properties, such as the preferred size of a window, are referred to as *hints*, since the window manager makes no guarantee that it will honor them.

host access list
See *access control list*.

hotspot A cursor has an associated *hotspot* that defines the point in the cursor which corresponds to the coordinates reported for the pointer.

identifier
Each server resource has an *identifier* or *ID*, a unique value that clients use to name the resource. Any client can use a resource if it knows the resource ID.

inferiors
The *inferiors* of a window are all of the subwindows nested below it: the children, the children's children, etc. The term *descendants* is a synonym.

inheritance (single vs. multiple)
Inheritance is the ability to obtain methods from a *superclass*. Multiple inheritance, which is not supported by Xt, would allow a class to obtain methods from multiple unrelated superclasses.

initialize method
The *initialize method* is invoked when a widget is created. It initializes instance variables and checks resource values. This method is part of the Core widget class.

initialize_hook method
The *initialize_hook method* is called just after the initialize method, and is responsible for initializing subpart instance variables and checking subpart resource values. This method is part of the Core widget class.

input focus
See *keyboard focus*.

InputOnly window
A window that cannot be used for graphics requests is called an *InputOnly* window. `InputOnly` windows are invisible and can be used to control such things as cursors, input event distribution, and grabbing. `InputOnly` windows cannot have `InputOutput` windows as inferiors.

InputOutput window
The normal kind of window that is used for both input and output is called an *InputOutput* window. It usually has a background. `InputOutput` windows can have both `InputOutput` and `InputOnly` windows as inferiors.

input manager
Control over keyboard input may be provided by an *input manager* client. This job is more often done by the window manager.

insert child method
A widget's *insert child method* is invoked when someone specifies it as a parent in an `XtCreateWidget` call. This method is part of the Composite widget class. This method is usually inherited.

instance An *instance* is a particular widget that has a class. An instance ID is returned by `XtCreateWidget`.

instantiate
When you *instantiate* a class, you create an instance of it.

Intrinsics
The X Toolkit defines functions and datatypes called *Intrinsics*.

key grabbing
A keyboard grab that occurs only when a certain key or key combination is pressed is called a *key grab*. This is analogous to button grabbing. Both are forms of passive grabs.

keyboard focus
The *keyboard focus* is the window that receives main keyboard input. By default the focus is the root window, which has the effect of sending input to the window that is being pointed to by the mouse. It is possible to attach the keyboard input to a specific window with `XSetInputFocus`. Events are then sent to the window independent of the pointer position.

keyboard grabbing
All keyboard input is sent to a specific window (or client, depending on `owner_events`) when the *keyboard* is *grab*bed. This is analogous to mouse grabbing. This is very much like a temporary keyboard focus window.

keycode A *keycode* is a code in the range 8-255 inclusive that represents a physical or logical key on the keyboard. The mapping between keys and keycodes cannot be changed, and varies between servers. A list of keysyms is associated with each keycode. Programs should use keysyms instead of keycodes to interpret key events.

keysym A *keysym* is a `#define`'d symbol which is a portable representation of the symbol on the cap of a key. Each key may have several keysyms, corresponding to the key when various modifier keys are pressed. You should interpret key events according to the keysym returned by `XLookupString` or `XLookupKeysym`, since this translates server-dependent keycodes into portable keysyms.

listener A *listener* style window manager sets the keyboard focus to a particular window when that window is clicked on with a pointer button. This is the window manager style used with the Apple Macintosh. This style is also referred to as click-to-type.

loose binding
A *loose binding* refers to the use of an asterisk (*) wildcard in a *resource specification* to indicate that the specification applies to zero or more hierarchy levels of widget instances or classes.

mapping
A window is said to be *mapped* if an `XMapWindow` or `XMapRaised` call has been performed on it. Unmapped windows are never viewable. Mapping makes a window eligible for display. The window will actually be displayed if the following conditions are also met:

1) All its ancestors are mapped.
2) It is not obscured by siblings.
3) The window manager has processed the request, if for a top-level window.

message In object-oriented programming, a *message* is used to invoke an object's methods. In Xt, you can think of the Intrinsics functions used to create, destroy, and set resource values as sending messages to widget classes.

method A *method* is a function internal to widget class that the Intrinsics invoke under specified conditions. Methods are provided as pointers to functions in widget class records.

modal pop up
A *modal pop up* is a pop-up widget that grabs the pointer when a mouse button is pressed in a particular window.

modeless pop up
A *modeless pop up* is a pop-up widget that doesn't grab the pointer on a mouse button.

modifier keys
Shift, Control, Meta, Super, Hyper, Alt, Compose, Apple, Caps Lock, Shift Lock, and similar keys are called *modifier keys*.

monochrome
A *monochrome* screen has only two colors: black and white. Monochrome is a special case of the `StaticGray` visual class, in which there are only two colormap entries.

non-maskable event
A *non-maskable event* is an event that is not selected by the application or widget via an event mask, but is instead sent automatically by the X server. The non-maskable events are `MappingNotify`, `ClientMessage`, `SelectionClear`, `SelectionNotify`, and `SelectionRequest`.

object In object-oriented programming, an *object* is a self-contained program unit containing both data and procedures.

object-oriented programming
Object-oriented programming is a way of defining classes and creating instances so that objects respond to messages with methods. One of its major benefits is the encapsulation of code.

obscure Window *A obscures* window *B* if *A* is higher in the global stacking order, and the rectangle defined by the outside edges of *A* intersects the rectangle defined by the outside edges of *B*. See *occlude*.

occlude Window *A occludes* window *B* if both are mapped, if *A* is higher in the global stacking order, and if the rectangle defined by the outside edges of *A* intersects the rectangle defined by the outside edges of *B*. The (fine) distinction between the terms *obscures* and *occludes* is that for *obscures*, the windows have to be mapped, while for *occludes* they don't. Also note that window borders are included in the calculation. Note that `InputOnly` windows never obscure other windows but can occlude other windows.

padding Some bytes are inserted in the data stream to maintain alignment of the protocol requests on natural boundaries. This *padding* increases ease of portability to some machine architectures.

parent A *parent* is a Composite widget instance specified as the parent argument in `XtCreateWidget`. Parents control a child's layout.

parent window
Each new window is created with reference to another previously-created window. The new window is referred to as the *child*, and the reference window as the *parent*. If *C* is a child of *P*, then *P* is the parent of *C*. Only the portion of the child that overlaps the parent is viewable.

passive grab
A *passive grab* is a grab that will become active only when a specific key or button is actually pressed in a particular window.

pixel value
A *pixel value* is an N-bit value, where *N* is the number of bit planes used in a particular window or pixmap. For a window, a pixel value indexes a colormap to derive an actual color to be displayed. For a pixmap, a pixel value will be interpreted as a color in the same way when it has been copied into a window.

pixmap A *pixmap* is a three-dimensional array of bits. A pixmap is normally thought of as a two-dimensional array of pixels, where each pixel can be a value from 0 to $(2^N - 1)$, where *N* is the depth (z-axis) of the pixmap. A pixmap can also be thought of as a stack of *N* bitmaps. A pixmap may have only one plane. Such a pixmap is often referred to as a bitmap, even though there is no bitmap type in X11.

plane When a pixmap or window is thought of as a stack of bitmaps, each bitmap is called a *plane*.

plane mask
Graphics operations can be restricted to affect only a subset of bit planes in a drawable. A *plane mask* is a bit mask describing which planes are to be modified.

pointer The *pointer* is the pointing device currently attached to the cursor, and tracked on the screens. This may be a mouse, tablet, track-ball, or joystick.

pointer grabbing
A client can actively *grab* control of the pointer, causing button and motion events to be sent to that client rather than to the client the pointer indicates.

pointing device

A *pointing device* is typically a mouse or tablet, or some other device with effective two-dimensional motion, that usually has buttons. There is only one visible cursor defined by the core protocol, and it tracks whatever pointing device is currently attached as the pointer.

pop up A *pop up* is a widget outside the normal parental hierarchy of geometry management. It is a child window of the root window that is popped up temporarily to give the user a piece of information, or to get some information from the user.

post-order traversal

When a tree structure is subject to *post-order traversal*, its root is visited last; for example, a function might process the left descendant, the right descendant, and then the root.

pre-order traversal

When a tree structure is subject to *pre-order traversal*, its root is visited first; for example, a function might process the root, and then the left and right descendant.

private header file

A *private header file* contains the internal class definitions of a widget. For example, the Label widget's internal class definitions are contained in the private header file *LabelP.h*. This file is included only in the widget *.c* file of this class and any subclasses, never in the application.

property

Windows may have associated *properties*, each consisting of a name, a type, a data format, and some data. The protocol places no interpretation on properties; they are intended as a general-purpose data storage and intercommunication mechanism for clients. There is, however, a list of predefined properties and property types so that clients might share information such as resize hints, program names, and icon formats with a window manager via properties. In order to avoid passing arbitrary-length property-name strings, each property name is associated with a corresponding integer value known as an atom. See also *atom*.

PseudoColor

PseudoColor is a visual class in which a pixel value indexes the colormap entry to produce independent red, green, and blue values. That is, the colormap is viewed as an array of triples (*RGB values*). The RGB values can be changed dynamically.

public header file

A *public header file* contains the declarations (e.g., class widget pointer) necessary to use a widget. For example, `labelWidgetClass` declarations are contained in the public header file *label.h*. This header file is included in the widget *.c* file and in the application source file.

quark A *quark* is an integer ID that identifies a string. In the context of Xt, this string is usually a name, class, or type string for the resource manager. Like atoms and resource IDs, quarks eliminate the need to pass strings of arbitrary length over the network. The quark type is `XrmQuark`, and the types `XrmName`, `XrmClass`, and `XrmRepresentation` are also defined to be `XrmQuark`.

query geometry method
Parents call `XtQueryGeometry` to invoke the child's *query geometry method* to find out a child's preferred geometry. This method is part of the Core widget class.

raise Changing the stacking order of a window so as to occlude all sibling windows is to *raise* that window.

real estate
The window management style characterized by the input being sent to whichever window the pointer is in is called *real-estate-driven*. This is the most common style of input management used in X.

realize method
The *realize method* creates a window on the display and is part of the Core widget class.

rectangle
A *rectangle* specified by `[x,y,w,h]` has an (infinitely thin) outline path with corners at `[x,y]`, `[x+w,y]`, `[x+w,y+h]`, and `[x,y+h]`. When a rectangle is filled, the lower-right edges are not drawn. For example, if `w=h=0`, nothing would be drawn when drawing the rectangle, but a single pixel when filling it. For `w=h=1`, a single pixel would be drawn when drawing the rectangle, four pixels when filling it.

redirect Window managers (or other clients) may wish to enforce window layout policy in various ways. When a client attempts to change the size or position of a window, or to map one, the operation may be *redirected* to the window manager, rather than actually being performed. Then the window manager (or other client that redirected the input) is expected to decide whether to allow, modify, or deny the requested operation before making the call itself. See also *window manager*.

reparenting
The window manager often *reparents* the top-level windows of each application in order to add a title bar and perhaps resize boxes. In other words, a window with a title bar is inserted between the root window and each top-level window. See also *save-set*.

reply Information requested from the server by a client is sent back to the client in a *reply*. Both events and replies are multiplexed on the same connection with the requests. Requests that require replies are known as round-trip requests, and, if possible, should be avoided since they introduce network delays. Most requests do not generate replies. Some requests generate multiple replies. Xt caches frequently-used data on the client side of the connection in order to minimize the need for round-trip requests. See Volume Zero, *X Protocol Reference Manual*, for a detailed discussion of the X Protocol.

representation type
A symbolic constant beginning with *XtR*, which is defined in *<X11/StringDefs.h>* or an application or widget header file, and which is used to define the data type of a resource. Xt automatically converts between resource settings, which are strings, and various representation types.

request A command to the server is called a *request*. It is a single block of data sent over the connection to the server. See Volume Zero, *X Protocol Reference Manual*, for a detailed discussion of the X Protocol.

resize method
The *resize method* of a widget is called when the widget's parent has changed its size. This method is part of the Core widget class. In composite widgets, this method calculates a new layout for the children. In simple widgets, this method recalculates instance variables so that the `expose` method can redraw the widget properly.

resource
1) A widget or application variable whose value can be set by the user or application programmer via the *Resource Manager*.
2) In the X Protocol and server documentation, windows, pixmaps, cursors, fonts, graphics contexts, and colormaps are known as *resources*. They all have unique identifiers (IDs) associated with them for naming purposes.

resource database
The *resource database* is the collection of possible places where resource specifications can be made. The most important of these are the *application-defaults file*, where the programmer specifies resource defaults, the `RESOURCE_MANAGER` server *property*, which the user can set with *xrdb*, and the *.Xdefaults* file in the user's home directory, where he or she specifies local resource preferences.

Resource Manager
The *Resource Manager* is a part of Xlib that merges a database consisting of several ASCII files, a server property, and values hardcoded by the application, and determines a unique value for each resource of each widget. It resolves conflicts between multiple settings for the same resource according to its internal precedence rules, and provides the widget or application with the resulting value.

resource setting
A *resource setting* is the actual value of a resource variable.

resource specification

A *resource specification* describes a "pathname" to a widget resource, using widget class and/or instance names, resource class or instance names, and optional wildcards. For example, the resource specification:

```
xbox.box.quit.label: Quit
```

specifies that the label resource of the quit widget, which is contained in the box widget in the *xbox* application, should be set to "Quit." The specification:

```
*Command.background: blue
```

which uses wildcards and widget classes, specifies that the background (color) resource of all Command widgets in any application should be set to "blue." It is up to the Resource Manager to resolve differences between resource specifications, which may have varying levels of specificity in the various locations that make up the resource database.

RGB values

See *colorcell*.

root

The *root* of a window, pixmap, or graphics context (GC) is the same as the root window of whatever drawable was specified in the call to create the window, pixmap, or GC. These resources can be used only on the screen indicated by this window. See *root window*.

root window

Each screen has a *root window* covering it. It cannot be reconfigured or unmapped, but otherwise acts as a full-fledged window. A root window has no parent.

round-trip request

A request to the server that generates a reply is known as a *round-trip request*. See *reply*.

save-set

The *save-set* of a client is a list of other clients' windows which, if they are inferiors of one of the client's windows at connection close, should not be destroyed, and which should be reparented and remapped if the client is unmapped. Save-sets are typically used by window managers to avoid lost windows if the manager should terminate abnormally. See *reparenting* for more background information.

scan line

A *scan line* is a list of pixel or bit values viewed as a horizontal row (all values having the same y coordinate) of an image, with the values ordered by increasing x coordinate values.

scan line order

An image represented in *scan line order* contains scan lines ordered by increasing y coordinate values.

screen A *screen* may not have physically independent monitors. For instance, it is possible to treat a color monitor as if it were two screens, one color and the other black and white or to have two separate color and monochrome screens controlled by one server. There is only a single keyboard and pointer shared among the screens. A `Screen` structure contains the information about each screen and the list is a member of the `Display` structure.

scrollbar
A *scrollbar* is an area on a window that allows a user to view different portions of a window's information by scrolling, or moving.

selection
Selections are a means of communication between clients using properties and events. From the user's perspective, a selection is an item of data which can be highlighted in one instance of an application and pasted into another instance of the same or a different application. The client that highlights the data is the owner, and the client into which the data is pasted is the requestor. Properties are used to store the selection data and the type of the data, while events are used to synchronize the transaction and to allow the requestor to indicate the type it prefers for the data and to allow the owner to convert the data to the indicated type if possible.

server The *server* provides the basic windowing mechanism. It handles IPC connections from clients, demultiplexes graphics requests onto the screens, and multiplexes input back to the appropriate clients. It controls a single keyboard and pointer and one or more screens that make up a single display.

server grabbing
The *server* can be *grabbed* by a single client for exclusive use. This prevents processing of any requests from other client connections until the grab is complete. This is typically a transient state for such tasks as rubber-banding, or to execute requests indivisibly.

set_values method
The Intrinsics invoke a widget's *set_values method* when one of the widget's resource values is changed. This method should return `TRUE` or `FALSE` to indicate whether the widget's `expose` method should be invoked.

set_values_almost method
The *set_values_almost method* is invoked when a parent rejects a widget's geometry request, but the parent sends back a compromise. This method is part of the Core widget class.

set_values_hook method
A widget can provide a *set_values_hook method* to allow the application to set resources of subparts.

sibling Children of the same parent window are known as *sibling* windows.

spring-loaded pop up
When a button press triggers a pop up, the pop up is called a *spring-loaded pop up*.

stacking order
Sibling windows may stack on top of each other, obscuring lower windows. This is similar to papers on a desk. The relationship between sibling windows is known as the *stacking order*. The first window in the stacking order is the window on top.

StaticColor
The *StaticColor* visual class represents a multiplane color screen with a predefined and read-only hardware colormap. It can be viewed as a degenerate case of `PseudoColor`. See *PseudoColor*.

StaticGray
The *StaticGray* visual class represents a multiplane monochrome screen with a predefined and read-only hardware colormap. It can be viewed as a degenerate case of `GrayScale`, in which the gray values are predefined and read-only. Typically, the values are linearly increasing ramps. See *GrayScale*.

stipple A *stipple* is a single plane pixmap that is used to tile a region. Bits set to 1 in the stipple are drawn with a foreground pixel value; bits set to 0, with a background pixel value. The stipple and both pixel values are members of the GC.

status Many Xlib functions return a *status* of `TRUE` or `FALSE`. If the function does not succeed, its return arguments are not disturbed.

subclass
A widget *subclass* has its own features plus many of the features of its superclasses. For example, since Composite is a subclass of Core, Composite has all the fields in Core plus its own unique fields. A subclass can inherit or replace most superclass features.

superclass
One widget is a *superclass* of a second widget when the second widget includes the first, or the Core widget is a *superclass* of the Composite widget because the Composite widget's definition depends on, or includes, the Core widget.

tight binding
A *tight binding* refers to the use of a dot (.) in a *resource specification* to indicate that the widget class or instance on the right side of the dot is a child of the widget whose class or instance name is on the left side of it.

tile A pixmap can be replicated in two dimensions to *tile* a region. The pixmap itself is also known as a *tile*.

time A *time* value in X is expressed in milliseconds, typically since the last server reset. Time values wrap around (after about 49.7 days). One time value, represented by the constant `CurrentTime`, is used by clients to represent the current server time.

top-level window
A child of the root window is referred to as a *top-level window*.

translation

A *translation* maps an event or event sequence into an action name. Once a translation is installed in a widget, the named action function will be invoked when the specified event seqeunce occurs in the widget. Translations are specified as ASCII strings.

translation table

A *translation table* lists one or more translations.

TrueColor

The *TrueColor* visual class represents a high-performance multiplane display with predefined and read-only RGB values in its hardware colormap. It can be viewed as a degenerate case of `DirectColor`, in which the subfields in the pixel value directly encode the corresponding RGB values. Typically, the values are linearly increasing ramps. See *DirectColor*.

type property

A *type property* is used to identify the interpretation of property data. Types are completely uninterpreted by the server; they are solely for the benefit of clients.

viewable

A window is *viewable* if it and all of its ancestors are mapped. This does not imply that any portion of the window is actually visible, since it may be obscured by other windows.

visible A region of a window is *visible* if someone looking at the screen can actually see it; that is, the window is viewable and the region is not obscured by any other window.

visual The specifications for color handling for a window, including visual class, depth, RGB/pixel, etc., are collectively referred to as a *visual*, and are stored in a structure of type `Visual`.

visual class

Visual class refers to `DirectColor`, `GrayScale`, `PseudoColor`, `StaticColor`, `StaticGray`, or `TrueColor`. It is a definition of the colormap type but not its depth.

widget The basic object in a toolkit, a *widget* includes both code and data, and can therefore serve as an input or output object.

window gravity

When windows are resized, subwindows may be repositioned automatically by the server, relative to an edge, corner, or center of the window. This attraction of a subwindow to some part of its parent is known as *window gravity*. Window gravity is a window attribute. See also *bit gravity*.

window manager

The user manipulates windows on the screen using a *window manager* client. The window manager has authority over the arrangement of windows on the screen, and the user interface for selecting which window receives input. See also *redirect*.

XYPixmap

The data for an image is said to be in *XYPixmap* format if it is organized as a set of bitmaps representing individual bit planes. This applies only to the server's internal data format for images. It does not affect normal programming with pixmaps.

ZPixmap

The data for an image is said to be in *ZPixmap* format if it is organized as a set of pixel values in scan line order. This applies only to the server's internal data format for images. It does not affect normal programming with pixmaps.

zoomed window

Some applications have not only a normal size for their top-level window and an icon, but also a *zoomed window* size. This could be used in a painting program (similar to the MacPaint™ fat bits). The zoomed window size preferences can be specified in the window manager hints.

Master Index

The Master Index combines Volumes Four and Five index entries, making it easy to look up the appropriate references to a topic in either volume. PM refers to the X Toolkit Intrinsics Programming Manual. RM refers to the X Toolkit Intrinsics Reference Manual.

Index

! (see modifiers)
directive (see translations)

A

accelerators about, PM:59, 152, 193, 211-217, 265, 531
adding, PM:212
and translations; conflicts with, PM:215;
translation table limitations, PM:213
compiling accelerator table, RM:274
defining default table in code, PM:216
event propagation, PM:213
for gadgets, PM:383
for menus, PM:379, 383
installing, PM:211; RM:242-244;
in multiple widgets, PM:215
not usable in gadgets, PM:386
(see also methods, display_accelerator; XtInstallAccelerators.)
accept_focus method PM:404; RM:387, 473
accepts_objects field PM:394
access control list PM:531
action hooks PM:408
actions PM:47, 49, 365, 369, 394, 411
about, PM:29, 32, 41-42, 531; RM:637
action proc format, PM:48
actions table; about, PM:531; XtActionProc, RM:374-375
adding to widget, PM:46
arguments to, PM:114
contrasted with callbacks, PM:49, 114
defined in widget implementation file, PM:148-149
gadget parent example, PM:393-394
in gadgets, PM:386
naming conventions, PM:45
passing arguments to, PM:114
registering with Translation Manager, RM:63-64
using event data, PM:229
widget instance pointer, PM:190
widget/application conflicts, PM:148
(see also XtAddActions; XtAppAddActions; XtMenuPopdown; XtMenuPopup.)
actions table adding, PM:46-47; RM:63-64, 81-82
declaring/registering with Resource Manager, RM:81-82
example, PM:48
format, PM:48
(see also actions.)
active grab PM:364
aliasing font names PM:457
AlmostProc RM:439
Alt key PM:202
(see also modifiers.)
ancestor PM:532
anonymous ftp PM:36
application applicationShellWidgetClass, RM:452-460
application contexts PM:102
about, PM:102, 410-413, 532
adding display, RM:188-189
creating, PM:102; RM:168

destroying and closing displays, RM:180
multiple, PM:411-412
XtOpenDisplay, PM:413
application resources about, PM:85
data structure, PM:86
retrieving values, PM:90
application-defaults file about, PM:34, 38, 253
directory, PM:38
naming conventions, PM:34, 38
ApplicationShell RM:429-430
applicationShellWidgetClass PM:284
appres utility PM:263
ArgList (see argument lists)
arguments Arg structure, PM:99
argc, PM:35, 96, 286
ArgsProc, RM:438
ARGSUSED lint comment, PM:45
argument lists, PM:57;
about, PM:99, 532;
constructing/modifying dynamically, RM:312-313;
creating, PM:57, 99-100;
creating dynamically, PM:100;
example, PM:99-100;
merging ArgList, RM:256;
XtMergeArgLists, RM:256
argument styles; command line, PM:95
argv, PM:35, 96, 286
to actions, PM:114, 125
varargs, PM:97
array allocating, RM:136
elements, determining number, RM:263
XtCalloc, RM:136
(see also XtNumber.)
aspect ratio PM:288
Athena widgets PM:19, 38, 69-70, 73, 75, 79, 278, 379
about, PM:17, 532
inheritance among, PM:19
(see also Box widget; Command widget; Dialog widget; Form widget; Grip widget; Label widget; MenuButton widget; Paned widget; scrollbars, Scrollbar widget; Text widget.)
atoms about, PM:303, 532
obtaining; example, PM:306
predefined, PM:305
standard, PM:314
augmenting translations PM:50-51

B

background about, PM:58, 532
background processing, RM:419-420
pixmap, PM:58
processing, PM:239
window attribute, PM:158
background option (-background, -bg), PM:92-93
backing store about, PM:532
backing_store window attribute PM:158-159
about, PM:269
binding tight vs. loose, PM:252
bit bit gravity window attribute, PM:158, 532
bit plane, PM:533
bitmap about, PM:293, 533
files, PM:106
BitmapEdit widget PM:70, 105-134, 181, 326
about, PM:106, 497
class hierarchy of, PM:139
instance part and record, PM:141
BitmapEditClassRec PM:140-141, 237
BitmapEditRec example, PM:141-142
bitwise OR combining mask symbols, RM:71, 120
Boolean values PM:265
border about, PM:533
border crossing events, RM:666-671
width, PM:322
border option (-border, -bd), PM:93
border window attribute PM:158
bordercolor option (-bordercolor), PM:93

borderwidth option (-borderwidth, -bw), PM:93
bounding box PM:175; RM:483
Box widget PM:21, 65-75, 212-217, 363-370
  example, PM:65-67, 69-70
BulletinBoard widget PM:429
ButtonPress event PM:201, 366; RM:651-653
ButtonRelease event PM:201, 366; RM:651-653
buttons PM:424
  grabbing, PM:533
  mapping, RM:689
  (see also command buttons.)
byte order PM:533

C

caching old size, PM:183
  resource, PM:271
  resource conversion, PM:271
  standard atoms, PM:314
  Xmu; initializing, PM:314
callbacks RM:376-377, 413
  about, PM:28, 32, 41-42, 533; RM:638
  adding, PM:43-44;
    more than one, PM:84;
    to callback list, RM:67-68;
    to callback resource, RM:65-66
  arguments to, PM:45
  callback list, PM:84;
    deleting method, RM:296-297;
    determining status, RM:233;
    executive methods, RM:133;
    popping down widget, RM:128-129;
    popping up widget, RM:124-127;
    XtCallbackExclusive, RM:124-125;
    XtCallbackNone, RM:126;
    XtCallbackNonexclusive, RM:127;
    XtCallbackPopdown, RM:128-129;
    XtCallCallbacks, RM:133;
    XtHasCallbacks, RM:233;
    XtRemoveCallback, RM:296;
    XtRemoveCallbacks, RM:297
  contrasted with actions, PM:49
  format, PM:45
  naming conventions, PM:45
  passing data, PM:81-83
  pop-up functions, PM:84
  procedure, RM:376-377
  XtAddCallback, RM:65-66
  (see also XtCallbackProc; XtTimerCallbackProc.)
Caption widget PM:429
cascading pop ups about, PM:357, 359, 374-379
  example, PM:376-377
case converter PM:205
  registering, RM:290
  XtRegisterCaseConverter, RM:290
chained methods (see inheritance)
change_managed method PM:322-324, 332; RM:474
  in constraint widgets, PM:349-350
CirculateNotify event RM:654
CirculateRequest event RM:654-655
class about, PM:18, 533
  class name; defined in Core class part, PM:152
  class part, PM:140;
    combining into class record, PM:140;
    lack of new fields, PM:141
  class record, PM:138;
    allocating storage, PM:142;
    BitmapEdit widget, PM:140-141;
    contents, PM:139
  class_initialize method, PM:156, 270
  extension stucture, PM:414
  hierarchy, PM:138-139;
    AT& T Open Look widgets, PM:363;
    Athena widgets, PM:156, 363;
    gadgets, PM:387-389;
    Motif, PM:363;
    (see also widget classes.)
  structure, PM:138-165;
    adding resource list to, PM:148
class_initialize method PM:156, 270, 339, 394, 415; RM:476

class_part_initialize method PM:156, 339, 342, 394; RM:477
client about, PM:6, 534
client-server model, PM:6
ClientMessage event RM:656-657
clipping region PM:534
color PM:58, 121, 159-160, 265-266, 290
determining available, PM:450
displaying, PM:450
false, PM:291
hexadecimal specification, PM:448
names, PM:447
RGB model, PM:449
specifying, PM:447
colorcell PM:450
about, PM:534
read-only, PM:451
read/write, PM:451
shared, PM:451
(see also colormap.)
colormap about, PM:58, 450
installing, PM:290
virtual, PM:290
window attribute, PM:159
(see also events, ColormapNotify.)
colormap about, PM:534
colormap window attribute PM:291
ColormapNotify event RM:656-658
command buttons PM:424, 426, 436
command line compiling, PM:36
options (see options)
Command widget PM:19, 39, 41-44, 49, 69, 79, 106, 212, 214-216, 365-371, 374, 378, 424-426, 436
compiling Xt PM:36
Composite XtOrderProc prototype procedure, RM:407
composite widgets about, PM:16, 321, 437, 534
as parent and child, PM:332
class, PM:16, 139;
about, PM:21, 65;
XtNinsertPosition resource, PM:335
general purpose, PM:429, 440
importance, PM:428
initial size, PM:325
inserting children, PM:335
management, PM:322
menus and control areas, PM:437
reasons for writing, PM:321
resources, PM:ix;
(see also resources.)
subclass; XtIsComposite, RM:246
using within other widgets, PM:322
(see also methods, insert_child.)
compound widgets PM:351-352
compression filters (see events, event filters)
ConfigureNotify event RM:659-660
ConfigureRequest event RM:661-662
connection PM:535
constants (defined) XtCacheAll, RM:638
XtCacheByDisplay, RM:638
XtCacheNone, RM:638
XtCacheRefCount, RM:638
Constraint destroy method, RM:481
initialize method, RM:496
set_values method, RM:513-514
constraint widgets PM:ix; RM:434-437, 434
about, PM:72, 321, 535; RM:434-437
class, PM:139;
about, PM:23;
part, PM:340;
refiguring child locations, PM:344-347
management, PM:336-351, 336
part structure, PM:338
resources, PM:73-74, 338; RM:434-437;
example, PM:73-74
writing, PM:336
XtIsConstraint, RM:247
(see also methods, Constraint destroy; methods, set_values; resources.)
consulting PM:525

containment PM:535
conventions action function names, PM:45
application-defaults file, PM:34, 38
callback function names, PM:45
font naming, PM:453
for XtInherit constants, PM:158
gadget internals, PM:388
instance variables order, PM:142
keysym naming, PM:199
public function names, PM:116
resource; class, PM:147;
names, PM:147, 248;
representation types, PM:147
widget; internals, PM:138;
source file names, PM:138;
structure declarations, PM:143, 164-165
conversion caching PM:271
converters PM:263-278, 411
color name to pixel value, PM:122, 265, 290
Convert.h, PM:273
explicitly invoking, PM:274-275
fontname to font, PM:265
function pointer to callback list, PM:270
int to Boolean, PM:266
int to Dimension, PM:266
int to float, PM:266
int to font, PM:266
int to pixel, PM:266
int to pixmap, PM:266
int to Position, PM:266
int to short, PM:266
int to unsigned char, PM:266
int to XColor, PM:266
Pixel to XColor, PM:266
registering, RM:69, 83-85;
case converter, PM:247; RM:290
string to bitmap, PM:269
string to boolean, PM:265
string to cursor, PM:265, 269
string to Dimension, PM:265
string to Display, PM:265
string to file descriptor, PM:265
string to float, PM:265
string to geometry, PM:265
string to initial state, PM:265
string to int, PM:265
string to orientation mode, PM:269
string to Position, PM:265
string to short, PM:265
string to translation table, PM:265
string-to-justify mode, PM:269
widget name to widget ID, PM:269
XColor to Pixel, PM:266
(see also XtAppAddConverter; XtSetTypeConverter.)
coordinate system PM:4, 535
Core PM:152
class part; fields, RM:484;
initializing, PM:151-153;
initializing example, PM:151-152
class structure; in gadgets, PM:388
Core widget class, PM:139; RM:438-442, 661-662;
about, PM:535
CoreClassPart structure, PM:141
CorePart structure, PM:141
fields, PM:152;
compress_enterleave, PM:153, 242;
compress_exposure, PM:153, 176, 242;
compress_motion, PM:153, 242;
display_accelerator, PM:152;
extension, PM:153;
superclass, PM:152;
tm_table, PM:153;
version, PM:153;
visible_interest, PM:153, 237;
widget_size, PM:152;
xrm_class, PM:153;
(see also methods, set_values_hook.)
instance record; height field, PM:325;
width field, PM:325
instance; setting size, PM:120
methods, RM:442
widget class, PM:17, 57;
about, PM:20;
class pointer, PM:120;
drawing into from application, PM:119-120;
hidden superclasses, PM:158;
instance default size, PM:120;

resources, PM:57;
superclasses, PM:387;
widgetClass class pointer, PM:120
(see also methods; methods, class_initialize; methods, class_part_initialize; methods, initialize.)
Core widget PM:17, 118, 197
drawing into, PM:119
updating, PM:122
counter incrementing inside XtSetArg, PM:55
CreateNotify event RM:663-664
Ctrl key PM:202
(see also modifiers.)
cursor about, PM:159, 265, 535
hotspot, PM:539
cursor window attribute PM:159
cut and paste (see selections)

D

data PM:546
data transfer completion method, RM:412
data types, RM:631
(see also structures.)
database obtaining for display, RM:179
(see also XtDatabase.)
DECNet PM:295
decoration PM:33, 293
default multiple click timing PM:207
default size PM:184
delete_child method PM:322; RM:478
depth PM:58, 173, 535
descendants PM:535
destroy method PM:157, 169, 187; RM:479
about, PM:187
Constraint, PM:340
example from BitmapEdit, PM:187
DestroyNotify event RM:665
destructors PM:271
details in translations (see translations)
device PM:536
dialog boxes PM:384
cascading, PM:384
grabs in, PM:384
without grabs, PM:384
Dialog widget PM:21, 75, 79, 351-352
popping up, PM:75
DirectColor PM:536
directories font, PM:452
display about, PM:6, 536
adding; XtOpenDisplay, RM:266-267
closing, RM:139
connecting to multiple displays, PM:413
depth, PM:450
display lists, RM:483
initializing, RM:188-189
lists, PM:175
XtCloseDisplay, RM:139
(see also XtDisplayInitialize.)
DISPLAY environment variable PM:58
display option (-display), PM:93
display pointer returning for widget, RM:187
display_accelerator method PM:216; RM:482
distributed processing PM:7
DoesBackingStore Xlib macro PM:159
DoesSaveUnders Xlib macro PM:159
double clicks PM:46, 206-207
downward chaining PM:154
drawing about, PM:121, 142, 153-155, 169, 175-178
after Expose event, PM:156
bitmap cells, PM:163
coordinate system, PM:4
due to set_values method changes, PM:180
in expose method, PM:170
into Core widget, PM:119-120
window attributes, PM:158-159
drop-down menu PM:359, 371
example, PM:373
vs. spring-loaded, PM:360

elements (see array)
encapsulation PM:31, 536
enter/leave compression PM:242
EnterNotify event PM:200-201, 242, 365-366; RM:666-671
EnterWindow event PM:222
environment variables DISPLAY, PM:58
XAPPLRESDIR, PM:253
XENVIRONMENT, PM:254
error database obtaining, RM:96-98, 205-206
error handling PM:96, 100, 399-402
and application contexts, PM:402
calling fatal error handler, RM:193-195
calling fatal error handler (high-level), RM:93-94
error resource database, PM:401; obtaining, RM:95; XtAppGetErrorDatabase, RM:95; XtAppGetErrorDatabaseText, RM:96-98; XtGetErrorDatabase, RM:205; XtGetErrorDatabaseText, RM:206
levels, PM:400
method, RM:394
procedure, RM:395-396
redefining handlers, PM:401
string conversion warnings, PM:402
XtAppError, RM:92
XtAppErrorMsg, RM:93-94
XtAppGetErrorDatabase, RM:95
XtAppGetErrorDatabaseText, RM:96-98
XtAppSetErrorHandler, RM:109
XtAppSetERrorMsgHandler, RM:110
XtError, RM:193
XtErrorMsg, RM:194-195
XtGetErrorDatabase, RM:205
XtGetErrorDatabaseText, RM:206
XtSetErrorHandler, RM:314
XtSetErrorMsgHandler, RM:315
errors error procedure
fatal error handler; calling (high-level), RM:93-94, 194-195
fatal error procedure; calling (low-level), RM:92; registering, RM:109-110, 314
listing, RM:703
nonfatal error procedure; calling, RM:116; registering, RM:114-115, 329
string conversion error message, RM:333
(see also XtAppError; XtAppErrorMsg; XtAppSetErrorHandler; XtAppSetErrorMsgHandler; XtAppSetWarningMsgHandler; XtError; XtErrorHandler; XtErrorMsg; XtErrorMsgHandler; XtSetErrorHand; XtSetErrorMsgHand; XtSetWarningMsgHandler.)
event handlers PM:32, 222-227
about, PM:30, 221, 536
adding, PM:222, 224-226; RM:398
dispatching, RM:186
for nonmaskable events, PM:225-226
procedure, RM:397-398, 400-401
raw, PM:227; registering raw, RM:76-77; removing raw, RM:302-303
reasons to use, PM:222
registering, RM:70-71
removing, RM:298-299
XtAddEventHandler, RM:70-71
XtAddRawEventHandler, RM:76-77
XtRemoveEventHandler, RM:298-299
XtRemoveRawEventHandler, RM:302-303
(see also XtDispatchEvent; XtEventHandler; XtInputCallbackProc.)
event masks about, PM:222
retrieving, RM:120-121
XtBuildEventMask, RM:120-121

event processing RM:107
XtAppProcessEvent, RM:107
events about, PM:10, 195
accessing specific data, RM:649
as argument of action, PM:49
border crossing, RM:666-671
ButtonPress, PM:201, 366; RM:651-653
ButtonRelease, PM:201, 366; RM:651-653
cancelling source, RM:301
CirculateNotify, RM:654
CirculateRequest, RM:654-655
ClientMessage, RM:656-657
ColormapNotify, RM:656-658
ConfigureNotify, RM:659-660
ConfigureRequest, RM:661-662
CreateNotify, RM:663-664
DestroyNotify, RM:663-665
dispatching handlers, RM:186
EnterNotify, PM:200-201, 242, 365-366
EnterWindow, PM:222
event compression, PM:536
event data; example of using in an action, PM:228
event filters, PM:153, 178, 242
event loop (see main loop)
event masks; about, PM:536-537; RM:632
event members; common, RM:650
event propagation, PM:537
event queue, PM:241
event sequences, PM:210
event source, PM:537
event structure, PM:227
event-driven programming, PM:10-11
event_mask window attribute, PM:159
expose, PM:10
Expose, PM:25, 116, 123, 125, 153, 156, 237, 242, 366; RM:672-673
FocusIn, PM:200, 222, 405, 537; RM:674-679
FocusOut, PM:200, 222, 405; RM:674-679
frozen event, PM:538
GraphicsExpose, PM:197, 226; RM:680-681
GravityNotify, RM:682
in action routines, PM:125
in gadgets, PM:386
input events; XtRemoveInput, RM:301
KeymapNotify, RM:683
KeyPress, PM:201, 406; RM:684-686
KeyRelease, PM:198, 201
LeaveNotify, PM:200-201, 242, 365-366; RM:666-671
list of types and structure names, PM:229-230
MapNotify, RM:687-688
MappingNotify, RM:689-690
MapRequest, RM:691
mocking up from action, PM:175; RM:483
MotionNotify, PM:201, 207, 222, 242, 366; RM:692-694
next event; returning, RM:262
NoExpose, PM:197; RM:680-681
nonmaskable, PM:195, 213, 224-226, 541;
example of handlers, PM:195, 213, 224
processing, RM:243
processing one event; XtProcessEvent, RM:283
propagation, PM:213
PropertyNotify, RM:695
ReparentNotify, RM:696
ResizeRequest, RM:697
returning next event, RM:103
selecting, PM:213
SelectionClear, PM:294, 304; RM:698
SelectionNotify, PM:294, 305, 310; RM:699
SelectionRequest, PM:294, 296, 305-306; RM:700
structures, RM:650
translation table abbreviations, PM:196-197
using inside actions or event handlers, PM:227-228; RM:375
VisibilityNotify, PM:153, 197; RM:701-702
XEvent; example, PM:228; RM:375;
union, RM:649
XtAppNextEvent, RM:103
XtAppPending, RM:106, 278

(see also exposure; XtDispatchEvent; XtMainLoop; XtNextEvent.)
expo.lcs.mit.edu PM:36
Expose event PM:366; RM:72, 672-673
Expose events PM:25, 169, 178; RM:485
expose method PM:154, 169, 175-178, 226; RM:483-486
about, PM:175
example from BitmapEdit, PM:175
in gadgets, PM:386, 391-392
exposure PM:10
compression, PM:176, 242; RM:484
XtAddExposureToRegion, RM:72
(see also events, Expose.)
extensions about, PM:12, 537
structures, PM:394, 414

F

fallback resources PM:257-258
fatal error (see XtError, XtErrorMsg)
fatal error handlers (see errors)
file events (see event handlers)
file input PM:230-233
registering file, RM:75, 86-87
source masks, PM:231
XtAddInput, RM:75
XtAppAddInput, RM:86-87
files filenames; character limit, PM:138
using names in resources, PM:265
floating point numbers PM:265
FocusIn event PM:200, 222; RM:674-679
FocusOut event PM:200, 222; RM:674-679
font conventions bolding, PM:vi
italics, PM:vi
typewriter font, PM:vi
font option (-font), PM:92-93
fonts about, PM:537
aliasing, PM:456
creating databases (mkfontdir), PM:458
directories, PM:452
display (xfd), PM:452
families, PM:452, 455
fonts.dir files, PM:457
naming convention, PM:453
printer, PM:452
screen, PM:452
specifying, PM:447
specifying as resources, PM:265
using file name as alias, PM:457
wildcarding, PM:454
foreground PM:537
foreground option (-foreground, -fg), PM:93
Form widget PM:21, 106, 336-351, 440
about, PM:72-74, 73, 429, 440
example, PM:440
layout method, PM:340
freeing storage block (see storage block)
ftp PM:36

G

gadgets RM:424
about, PM:157, 357
accelerators, PM:383; not usable, PM:386
actions in, PM:386
class hierarchy, PM:387-388
class structure, PM:389
composite parent, PM:386, 391-395
Core class structure, PM:388
drawbacks of, PM:386
event handling in, PM:386
expose method, PM:391
implementation file, PM:390
instance structure, PM:390
internals, PM:388-391
private header file, PM:389-390
public header file, PM:391
query_geometry method, PM:391
reason for, PM:385
set_values_almost method, PM:391
Sme, PM:379-391
SmeBSB, PM:379-391
SmeLine, PM:379-391
superclass, PM:390

unused Core fields in, PM:390-391
games PM:234
GCs (see graphics contexts)
geometry management PM:69-70, 287, 322, 335; RM:431-433
about, PM:35, 181, 183-186, 321-352, 538; RM:641
border width, PM:322
changes, RM:244-246
changing (XtMakeGeometryRequest), PM:325; RM:244-246
composite resource, PM:335
compound widgets, PM:351-352
constraint class part, PM:340
constraint management, PM:336
delaying recalculation, PM:351
height, PM:325
initial geometry negotiation, PM:324
inserting children, PM:335
minimal useful size, PM:330
querying, RM:284-286;
preferred geometry, PM:332-333;
XtQueryGeometry, RM:284-286
resizing, PM:321
scope, PM:322
size preferences, PM:333
stacking order, PM:322, 352
trickle-down, PM:333
unmanaging widget, PM:332
width, PM:325
(see also methods; XtDestroyWidget.)
geometry option (-geometry), PM:93
geometry_manager method RM:487
get_values_hook method PM:157, 216, 279, 479; RM:225, 491-492
global variables PM:79, 82
glyph PM:538
grabs about, PM:363, 531, 538; RM:641
active vs. passive, PM:364
adding or removing explicitly, PM:385
exclusive vs. nonexclusive, PM:364, 377
global, PM:364
grab modes, PM:377
in dialog boxes, PM:384
keyboard, PM:363
passive, PM:364, 542
pointer, PM:363
reasons for in menus, PM:365
XtAddGrab, RM:73-74
XtRemoveGrab, RM:300
graphics PM:121
graphics primitive, PM:538
(see also graphics contexts.)
graphics contexts about, PM:121, 142, 159, 538
caching, PM:170, 173
changing, PM:173, 180
creating, PM:170, 172-174
deallocating, RM:293
destroying, RM:181
exclusive or logical function, PM:302
freeing, PM:180, 187; RM:181
hardcoding values in, PM:174
obtaining, RM:207-208
read-only, PM:173
reasons for, PM:121
setting with resources, PM:174
(see also XtDestroyGC; XtGetGC; XtReleaseGC.)
GraphicsExpose event RM:72, 680-681
GraphicsExpose events PM:226
gravity PM:538
GravityNotify event RM:650, 682
GrayScale PM:538
Grip widget PM:20, 70

H

hardcoding resources, PM:40, 97, 99
translations, PM:51
header files PM:32, 54, 59, 231, 387, 464
not included twice, PM:141
private, PM:543
public, PM:543
height PM:325
checking in initialize method, PM:170

hexadecimal color specification PM:448
hints about, PM:284-294, 538; RM:682
icon position, PM:287
position, PM:62
size, PM:287
size increment, PM:287
hooks RM:225, 327
action, PM:408
host access list PM:538
Hyper key PM:202
(see also modifiers.)

I

ICCCM PM:11, 283, 303, 306, 312-317
icon creating; pixmap, PM:292-293
pop-ups, PM:286
setting; name, PM:292; pixmap, PM:283, 292
starting application as, PM:287
icon size and position PM:460
iconic option (-iconic), PM:93
identifier PM:539
ifndef statement PM:141
implementation file (see widget)
include files PM:32, 54, 59, 138, 141
in widget implementation, PM:145
incremental selections PM:317
inferiors PM:539
inheritance about, PM:19, 52, 59, 151-160, 539; RM:682
adding features to superclass, PM:160
among Athena Widgets, PM:19
among Motif widgets, PM:419
among Open Look widgets, PM:419
in widget class and instance record, PM:139-140
of AT& T, PM:425
of chained methods, PM:154
of conflicting methods, PM:160
of Core resources, PM:57, 59
of self-contained methods, PM:154
of superclass method, PM:157
resources, PM:145, 147-148
styles, PM:154-155
using XtInherit constants, PM:154
widget not using resource value, PM:62
initial size PM:325
initialize method PM:169-174, 306, 328, 480; RM:493-495
about, PM:170
Constraint, PM:341
creating GCs in, PM:172
example from BitmapEdit, PM:172
initialize_hook method PM:279
input from file, PM:231-233
from pipe, PM:233-234
input events method, RM:397-398
input focus, PM:539
input manager, PM:539
input source masks, PM:231
InputOnly window, PM:539
InputOutput window, PM:155, 539; RM:672
input queue determining events, RM:106, 278
examining head, RM:104-105, 277
XtAppNextEvent, RM:103
XtAppPeekEvent, RM:104-105
XtAppPending, RM:106, 278
XtPeekEvent, RM:277
insert_child method PM:322; RM:499
insert_position RM:407
instance about, PM:18, 539
record, PM:138;
adding variables to, PM:142;
allocating storage, PM:142;
BitmapEdit widget, PM:141-142;
contents, PM:139
structures, PM:138-165, 170;
constraints in, PM:338
Inter-Client Communication Conventions Manual (see ICCCM)
Intrinsics PM:145
about, PM:9, 540
intrinsic classes; introduction to, RM:423-428
Intrinsic.h header file, PM:32

IntrinsicP.h header file, PM:145
selection timeout, RM:112

K

key events (see events or translations)
key translation registering, RM:318
XtKeyProc, RM:402-404
XtSetKeyTranslator, RM:318
keyboard (see also accept_focus.)
keyboard focus PM:289; RM:473
about, PM:288, 404-405
accept/reject method, RM:387
redirecting input, RM:316-317
setting, PM:289
styles, PM:289, 540
window, RM:674
(see also accept_focus; XtNinput resource; XtSetKeyboardFocus.)
keyboard grabbing (see grabs)
keyboard mapping RM:689
keyboard shortcuts (see accelerators)
keycodes about, PM:198, 406, 540
translating; keycode-to-keysym, RM:337-340; XtTranslateKey, RM:337-338; XtTranslateKeycode, RM:339-340
KeymapNotify event RM:683
KeyPress event PM:201
KeyRelease event PM:201; RM:684-686
keysyms about, PM:198, 406, 540
converting case, PM:205; RM:379-380
determining case, RM:167
key generated, PM:203
keysymdef.h header file, PM:199, 202
naming conventions, PM:199
XtCaseProc, RM:379-380
XtConvertCase, RM:167

L

Label widget PM:19, 30, 75, 118, 181, 248, 323, 366, 464
Label.h file, PM:31
labels setting, PM:53
language portability PM:52
layout Form method PM:340
LeaveNotify event PM:200-201, 242, 365-366
LeaveWindow event PM:222, 374, 378
lint PM:45
List widget PM:20, 436
class, PM:20, 222
loose bindings PM:252, 540

M

macros PM:403-404
main loop PM:32, 36
customizing example, PM:241
internals, PM:241
XtAppMainLoop, RM:102
XtMainLoop, RM:243
MapNotify event RM:687-688
mapping PM:197
about, PM:59, 540
button, RM:689
keyboard, RM:689
pop-up shell; XtPopup, RM:280-281
unmapping pop-up shell; XtPopdown, RM:279
widgets; XtMapWidget, RM:252
(see also MappingNotify event; widget, mapping.)
MappingNotify event RM:689-690
MapRequest event RM:691
mechanism without policy PM:10, 283
memory allocation PM:408
for widget instance record, PM:152
menus about, PM:36, 357-395, 428
accelerators in, PM:379, 383
cascaded, PM:359, 374-379
drop-down, PM:359, 371-373; example, PM:372-373
operation of, PM:361

panes in, PM:361
pointer grabbing, PM:363, 365
popping down, PM:363, 371, 379;
MenuPopdown, RM:253
popping up, PM:369;
MenuPopup, RM:254-255
popping up with callbacks, PM:371
pull-down, PM:359
SimpleMenu widget, example, PM:380, 382
spring-loaded, PM:359, 363
XtMenuPopdown action, PM:365
XtMenuPopup action, PM:365
messages about, PM:31, 541
OOP vs. Xt, PM:31
Meta key PM:202
methods about, PM:155-157, 541
accept_focus, PM:156, 404-405; RM:473;
about, PM:531
and instance structure, PM:139
change_managed, PM:322-324, 332, 349; RM:474;
about, PM:533;
in constraint widgets, PM:349-350
class_initialize, PM:156, 270, 394; RM:476;
about, PM:534
class_part_initialize, PM:156; RM:477
Constraint; destroy, RM:481;
initialize, PM:342; RM:496
Constraint set_values, RM:513-514
constraint widgets; destroying, PM:340;
initializing, PM:341;
resizing, PM:347
constraint_destroy, RM:437
declarations in widget implementation file, PM:150
delete_child, PM:322, 335; RM:478;
about, PM:535
destroy, PM:157, 169, 187, 340; RM:479;
about, PM:536;
example, PM:187
display_accelerator, PM:216; RM:482
drawing; due to changes in set_values, PM:180;
in expose, PM:170
expose, PM:31, 156, 169-170, 175-178, 226, 537; RM:484-485;
in gadget parent, PM:391-392;
in gadgets, PM:386, 391;
XtExposeProc, RM:438, 483-486
Form layout, PM:340
gadget; expose, PM:391;
query_geometry, PM:391;
set_values_almost, PM:391
geometry_manager, PM:322-323, 343; RM:487
get_values_hook, PM:157, 216, 279, 479; RM:491-492, 491, 517;
about, PM:538;
example, PM:279; RM:491
in OOP, PM:31
inheritance; adding to superclass, PM:160;
of superclass, PM:157
initialize, PM:155, 169-174, 170, 341; RM:493-495;
about, PM:539;
calling XInternAtom, PM:169-174, 170, 328;
example, PM:170, 172;
in constraint widget, PM:341
initialize_hook, PM:155, 279;
about, PM:539
insert_child, PM:322; RM:491, 499;
about, PM:539
insert_position, PM:335; RM:499
layout Form, PM:340
not known to Xt, PM:340
query_geometry, PM:157, 169, 184-186, 322; RM:501;
about, PM:169, 184-186, 322, 544;
in constraint widget, PM:329-330, 351;
in gadgets, PM:350, 391
realize, PM:156, 288, 328; RM:505-507;
about, PM:544
reconciliation, PM:160

resize, PM:156, 169, 181-183, 322, 325, 347; RM:508;
about, PM:545;
in composite widget, PM:329;
in constraint widget, PM:347-348;
in gadget parent, PM:393
resources and set_values, PM:178-181
set_values, PM:156, 169-170, 178-181, 328-329, 349; RM:510-512;
about, PM:547;
redrawing, RM:434
set_values_almost, PM:334-335; RM:515-516;
about, PM:547;
in gadgets, PM:156, 391
set_values_hook, PM:156, 279-280; RM:517-518, 517;
about, PM:547
minimal useful size PM:330
mkfontdir PM:458
Mod keys PM:202
(see also modifiers.)
modal cascade PM:360
modal pop ups PM:360
modifiers !, PM:205
adding, PM:203
and event sequences, PM:207
case-specifics, PM:205
colon, PM:205-206
displaying list, PM:203
for button events, PM:208
keys, PM:201-206, 541; RM:712-719
matching exactly, PM:205
negating, PM:204
None, PM:205
monochrome PM:541
Motif PM:361, 419, 422
motion compression PM:242
MotionNotify event PM:201, 222, 242, 366; RM:692-694
multiple click timing PM:207
multiple toplevel shells PM:413

N

name option (-name), PM:93
naming conventions widgets, PM:463
newlines in translations, PM:194
nonfatal error PM:197
registering method, RM:115
registering procedure, RM:329, 331
(see also error; error handling.)
nonmaskable events (see events)
non-widget objects RM:423
notify method PM:394
notify modes (see translations)

O

object RM:443-446
about, PM:30, 541
Object class, PM:387; RM:423
object-oriented programming, PM:541
objectClass RM:444
ObjectClassPart RM:443
ObjectClassRec RM:444
object-oriented programming PM:ii, 30-31, 226
ObjectPart RM:444
ObjectRec RM:445
obscure PM:541
occlude PM:541
OOP (see object-oriented programming)
optimization PM:125, 183, 411
options abbreviating, PM:92
argument styles, PM:95
-background, PM:92-93
-background (-bg), PM:447
-bd, PM:93
-bg, PM:92-93
border (-bd), PM:447
-bordercolor, PM:93
-borderwidth, PM:93
-bw, PM:93
command line, PM:91; styles, PM:95
custom, PM:254
defining your own, PM:93
-display, PM:93
-fg, PM:93
-font, PM:93
-foreground (-fg), PM:447
-geometry, PM:288, 458-460
handling errors in, PM:96
-iconic, PM:93, 287

options table; example, PM:94; xbitmap, PM:108
overriding standard, PM:96
-rv/+rv, PM:93
-sectionTimeout , PM:93
-selectionTimeout, PM:93
standard, PM:91-92
-synchronous, PM:93
-title, PM:93
-xrm, PM:91, 254
OR operator RM:71, 120, 298, 302
OverrideShell RM:447
OverrideShell widget class, PM:286; RM:452-460
overrideShellWidgetClass, PM:79
overriding override_redirect window attribute, PM:159
redirect, PM:286
standard options, PM:96
translations, PM:50-51

P

padding PM:542
Paned widget PM:20, 70
parent about, PM:542
parent window, PM:542
parsing command-line arguments, PM:93-96
translations, PM:51
part (vs. record) PM:140
passive grab PM:364
path resources PM:255
pipe input PM:233-234
pixel values about, PM:121, 542
pixmap about, PM:58, 175, 542; RM:483
freeing, PM:187
icon, PM:292
updating in widget, PM:118
XYPixmap format, PM:549
ZPixmap format, PM:550
plane PM:542
plane mask, PM:542
(see also bit plane.)
pointer PM:363, 542
about, PM:3
dereferencing, PM:535
pointer events (see events or translations)
pointer grabbing; about, PM:542
pointing device, PM:542
(see also grabs.)
pop ups about, PM:17, 75, 543
cascading; about, PM:357
creating in work procedure, PM:240; RM:420
creating just before popping up, PM:81
from callback function, PM:79
linking group, PM:286
modal, PM:360;
about, PM:541
modeless, PM:360;
about, PM:541
moving to desired position, PM:80
OverrideShell, PM:286
sensitivity, PM:385
spring-loaded, PM:360, 547
using, PM:75-76, 79
when application is iconified, PM:286
pop-up menus PM:357
example, PM:366, 369
(spring-loaded) using Box widget, PM:366, 369-370
pop-up shell creating, RM:171-173
mapping, RM:280-281
unmapping, RM:279
(see also XtCreatePopupShell.)
portability PM:99, 231, 389, 407, 410-411
position about, PM:62, 265
hints, PM:287
relative to root window, PM:294
setting with resources, PM:61
PRIMARY selection PM:296
printer fonts (see fonts)
private header file BitmapEdiP.h, PM:138-143
private instance variables PM:142
process input RM:102
XtAppMainLoop, RM:102
program structure PM:32
properties about, PM:294, 543
and atoms, PM:303
PropertyNotify event RM:695
protocol PM:5

PseudoColor PM:543
public functions PM:161
about, PM:107
naming conventions, PM:116
reasons to use, PM:109
public header file PM:32, 391
BitmapEdit.h, PM:138, 160-162
public instance variables (see resources)
public routines PM:351
pull-down menu PM:359

Q

quarks PM:276, 543
query_geometry method PM:157, 184-186, 391; RM:501
about, PM:184
example from BitmapEdit, PM:186
gadgets in, PM:391
in gadgets, PM:350-351

R

raise PM:544
raw event handlers PM:227; RM:76-77
(see also events.)
realization PM:35
about, PM:176, 180
realize method PM:154, 159, 288, 328, 480; RM:505-507
about, PM:157
RealizeProc RM:438
rectangle PM:544
RectObj class PM:158, 387, 390; RM:423, 448-451
RectObjClass RM:450
RectObjClassPart RM:449
RectObjClassRec RM:450
RectObjPart RM:450
RectObjRec RM:450
redrawing windows PM:25, 175
reference pages list, RM:617
regions PM:178, 392; RM:72, 485
registering callbacks, RM:65-66; list, RM:67-68
converters, PM:269-274; RM:69, 83-85, 290
event handlers, PM:224-226; RM:70-71
fatal error condition procedure, RM:110, 314
fatal error procedure, PM:156, 399-402; RM:109
file, RM:75, 86-87
nonfatal error condition procedure, RM:114, 315, 329, 331
nonfatal error procedure, PM:399-402; RM:115
raw event handler, RM:76-77
work procedure, RM:79, 89
registering/declaring actions (see actions)
Release 4 (R4) PM:i, 36, 116, 200, 289, 357, 359, 385, 414
compared to R3, PM:467-493
initialize_hook and set_values_hook obsolescence, RM:491, 497
removing callbacks, RM:295-297
grabs, RM:300
input, RM:301
raw event handlers, RM:302-303
timeouts, RM:304
reparenting PM:544
about, PM:293, 373
(see also window manager.)
ReparentNotify event RM:696
reply PM:544
representation type PM:86, 88
about, PM:247, 249, 545
resize method PM:169, 181-183, 322; RM:508
about, PM:181
example from BitmapEdit, PM:183
in gadgets, PM:393
ResizeRequest event RM:697
resizing PM:322
about, PM:69, 169, 181-183, 321
caching old size, PM:183
parent widget, RM:508
(see also events, ResizeRequest; methods, resize.)
resource conversion, RM:184
resource database obtaining for display, RM:179
(see also XtDatabase.)

resource list copying; from ArgList, RM:323-324
copying to argument list, RM:223-224
copying; (see also XtGetSubvalues; XtSetSubvalues.)
default values; retrieving, RM:211-212
retrieving; (see also XtGetResourceList.)
updating, RM:201-203;
by name or class, RM:221-222
(see also XtGetSubResources; XtGetApplicationResources.)
Resource Manager (see actions)
resources about, PM:23-24, 34, 38, 52, 57, 59, 248-251, 545; RM:431-432
and set_values method, PM:178-181
application, PM:85
caching, PM:271
changing value, PM:178
checking validity, PM:170
classes and instances, PM:88
comment character in files, PM:253
constraint, PM:73; RM:434-437
copying; from ArgList to widget, RM:327-328;
from widget to argument list, RM:225-226;
XtGetValues, RM:225-226;
XtSetValues, RM:327-328
default address; interpreting, PM:251
default value; converting, PM:267;
setting, PM:88, 267
defined by Core, PM:57
defined in widget implementation file, PM:145, 147-148
defining characteristics, PM:88-89
defining in widget, PM:145, 147-148
fallback, PM:258
Form widget, PM:73, 337
format; of definitions, PM:252-253
getting from application, PM:52
hardcoding advantages, PM:40
in instance record, PM:142
inheritance of, PM:52, 145, 147-148
interactions between, PM:60, 62;
example, PM:60
loading; from .Xdefaults file, PM:60;
with xrdb, PM:60
looking up values, PM:251-263
loose bindings, PM:252
name, PM:88
naming conventions, PM:248
precedence rules, PM:260-263
representation types, PM:86, 88, 249
resource conversion, PM:88;
converters in Xmu, PM:266;
method, RM:408;
registering, PM:270;
XtConverter, RM:383-386;
(see also XtResourceDefaultProc.)
resource database, PM:545;
sources, PM:253
resource file format, RM:709
resource list; declaring, PM:248;
example, PM:87-88;
format, PM:88-89
Resource Manager, PM:545
resource setting, PM:545
resource value; getting, PM:57
retrieving; application resources, PM:90-91;
resource default at run-time, PM:268
RESOURCE_MANAGER property, PM:254
setting; for multiple widgets, PM:68;
for widget hierarchy, PM:68;
in application, PM:52;
with XtSetValues, PM:54;
XtNinput, PM:290
size, PM:251
sources priority, PM:254
specifications, PM:260-263, 545;
errors in, PM:39;
format, PM:39-40;
merging of duplicate, PM:253-255;
wildcards in, PM:252
symbolic constants, PM:40, 54
tight bindings, PM:252
type conversion, PM:40, 263-278

type converter, XtConvert, RM:142-143
types table, PM:249
use of classes, PM:249
XtN constants, PM:160
XtNaccelerators (Core), PM:58
XtNallowShellResize (Shell), PM:286
XtNancestorSensitive (Core), PM:58
XtNargc (Shell), PM:285-286
XtNargv (Shell), PM:285-286
XtNbackground (Core), PM:58
XtNbackgroundPixmap (Core), PM:58
XtNbaseHeight (Shell), PM:286-287
XtNbasewidth (Shell), PM:286
XtNbaseWidth (Shell), PM:287
XtNborderColor (Core), PM:58
XtNborderPixmap (Core), PM:58
XtNborderWidth (Core), PM:58
XtNcolormap (Core), PM:58
XtNcreatePopupChildProc (Shell), PM:285
XtNdefaultDistance (Form), PM:336
XtNdepth (Core), PM:58
XtNdestroyCallback (Core), PM:44, 58
XtNgeometry (Shell), PM:285-286, 288
XtNheight (Core), PM:58
XtNheightInc (Shell), PM:286-287
XtNhorizDistance (Constraint), PM:336
XtNiconic (Shell), PM:285, 287
XtNiconMask (Shell), PM:286
XtNiconName (Shell), PM:286, 292
XtNiconPixmap (Shell), PM:286, 292
XtNiconWindow (Shell), PM:286
XtNiconX (Shell), PM:285, 287
XtNiconY (Shell), PM:285, 287
XtNinitialState (Shell), PM:286
XtNinput (Shell), PM:286, 288-290
XtNinsertPosition (Composite), PM:335
XtNmappedWhenManaged (Core), PM:58, 332
XtNmaxAspectX (Shell), PM:286-287
XtNmaxAspectY (Shell), PM:286-287
XtNmaxHeight (Shell), PM:286-287
XtNmaxWidth (Shell), PM:286-287
XtNminAspectX (Shell), PM:286-287
XtNminAspectY (Shell), PM:286-287
XtNminHeight (Shell), PM:286-287
XtNminWidth (Shell), PM:286-287
XtNoverrideRedirect (Shell), PM:285-286
XtNpixmap (Core), PM:118
XtNpopdownCallback (Shell), PM:371
XtNpopupCallback (Shell), PM:371, 373
XtNsaveUnder (Shell), PM:286
XtNscreen (Core), PM:58
XtNsensitive (Core), PM:58
XtNtitle (Shell), PM:285, 292
XtNtransient (Shell), PM:285-286
XtNtranslations (Core), PM:49, 51, 58, 114
XtNvertDistance (Constraint), PM:336
XtNwaitForWm (Shell), PM:285-286
XtNwidth (Core), PM:58
XtNwidthInc (Shell), PM:286-287
XtNwindowGroup (Shell), PM:285-286
XtNwmTimeout (Shell), PM:285-286
XtNx (Core), PM:58
XtNy (Core), PM:58
(see also resource list.)
resources conversion XtDirectConvert, RM:184
reverse option (-rv/+rv) PM:93

RGB color model, PM:449-450
root window PM:58, 79, 288, 294, 369, 404, 546
root_geometry_manager PM:479
root_geometry_manager method RM:455
round trip request PM:32
rubber-band outline PM:181, 302

S

save_under window attribute PM:159
saving under PM:159
scan line PM:546
screen PM:3
about, PM:6, 546
example of typical X Window, PM:5
layout of, PM:287
monochrome, PM:173
screen fonts; (see fonts)
specifying default, PM:413
scrollbars about, PM:15, 68, 429, 441, 547
adding to application, PM:111; example, PM:111
adding to xbitmap, PM:111
and Expose events, PM:116
calculating thumb size, PM:115-117; example, PM:115-117
creating, PM:113
Scrollbar widget, PM:15, 20, 44, 70, 110-113, 269
selection data RM:185, 269-270 (see also XtOwnSelection.)
selection timeout (see timeouts)
SelectionClear event RM:698
SelectionNotify event RM:699
SelectionRequest event RM:700
selections PM:198, 283, 294-318, 294, 547
adding to clipboard, PM:316
and CurrentTime, PM:304
asserting ownership, PM:303-304
converting; example, PM:308-309, 315-316; selection, PM:307-309; standard selections, PM:314-316; RM:390, 392
deleting, PM:316
disowning, PM:318
handling large selections, PM:316; RM:392
highlighting, PM:298, 302-303; example, PM:299, 302-303
incremental, PM:317
losing selection, PM:304, 312
pasting, PM:307; example, PM:310-311; selection, PM:310-311
querying for desired target, PM:313-317
requesting selection, PM:304-305
selection data; method, RM:409-410; obtaining, RM:215; obtaining in multiple formats, RM:218-219; (see also XtGetSelectionValue; XtGetSelectionValues.), XtConvertSelectionProc, RM:389-392; XtDisownSelection, RM:185
selection method XtSelectionDoneProc; (see also XtSelectionCallbackProc; XtLoseSelectionProc; XtSelectionDoneProc.)
selection timeout (see timeouts)
setting timeout, PM:318
target types, PM:305-307
XA_CLIPBOARD, PM:303, 316-317
XA_MULTIPLE, PM:316; RM:392
XA_PRIMARY, PM:296, 303
XA_SECONDARY, PM:303
XA_TARGETS, PM:306, 313
(see also events, SelectionRequest; SelectionClear event; SelectionNotify event; SelectionRequest event.)
self-contained methods (see inheritance)
sensitivity PM:59, 81, 99, 243
checking state; XtIsSensitive, RM:233
in pop-up callbacks, PM:385

setting state; XtSetSensitive, RM:322
server about, PM:6, 547
round trip request, PM:546
server code; guide to, PM:527
server grabbing, PM:547
server request, PM:545
server resources; freeing, PM:187
set_values (see also XtSetValues.)
set_values method PM:156, 169-170, 178-181, 328-329; RM:510-512
about, PM:178
example from BitmapEdit, PM:179
set_values_almost method PM:391; RM:515-516
in gadgets, PM:156
set_values_hook method PM:156, 279-280; RM:517-518
Shell PM:254; RM:452-460
Shell types; ApplicationShell, RM:453;
OverrideShell, RM:453;
Shell, RM:453;
TopLevelShell, RM:453;
TransientShell, RM:453;
VendorShell, RM:453;
WMShell, RM:453
Shell widget, PM:17, 65, 106;
about, PM:21
Shell widget class; about, PM:17, 34, 61, 65, 70, 283
ShellClassPart, RM:454-456
ShellPart, RM:457-459
(see also environment variables.)
Shell widget class RM:517
about, RM:452-460
reason for invisibility, PM:67
resources, PM:285-291, 293-294
root_geometry_manager method, RM:455
subclass; XtIsShell, RM:234
types, RM:453
XtNbasewidth, PM:286
sibling PM:547
Simple widget PM:19, 420
SimpleMenu widget PM:361, 379-383, 391-395
single-line input field PM:294
sink in Athena Text widget, PM:278
size PM:321
hints, PM:287
preferences, PM:333
sizeof, PM:152
Sme gadgets PM:379-391
software architecture PM:8
source code; obtaining, PM:36
files for widget, PM:138
in Athena Text widget, PM:278
source compatibility PM:414
spring-loaded menu PM:363-371
vs. drop-down, PM:360
spring-loaded pop up (see pop ups)
stacking order PM:322, 352, 547
standard colormaps Xmu utilities, PM:291
StaticColor PM:548
StaticGray PM:548
status PM:548
stdio.h file PM:145
stipple PM:548
storage allocating, RM:248;
for data type, RM:260;
XtNew, RM:260
XtMalloc, RM:248
storage block freeing, RM:198
resizing; XtRealloc, RM:289
(see also XtFree.)
string copying; XtNewString, RM:261
error message; XtStringConversionWarning, RM:333
StringDefs.h header file, PM:88; RM:721
string to bitmap converter PM:269
StringDefs.h header file PM:32, 54, 145, 269
strings setting, PM:53
StringToWidget resource converter PM:73
structure determining field's byte offset, RM:264
(see also XtOffset.)

structure of Xt applications PM:32
subclass PM:19
about, PM:548
submenus PM:377
cascading, PM:379
creating, PM:376
popping up, PM:84, 375
(see also XtPopup.)
subparts PM:152, 157, 278-280
subresources PM:152, 157, 278-280
managing, PM:279-280
Super key PM:202
(see also modifiers.)
superclass PM:152, 179
about, PM:19, 548
class structure, PM:140
gadget, PM:390
inheritance, PM:59
of Core, PM:387
synchronous option (-synchronous/+synchronous), PM:93
syntax functions PM:96

T

TCP/IP PM:295
Text widget PM:20-21, 79, 153, 294, 433, 443
tight bindings PM:252, 548
tilde PM:204
tiling PM:58
about, PM:548
time PM:548
timeouts about, PM:234
adding, PM:234
and visibility interest, PM:237
callback method, RM:413
example, PM:235-236
invoking procedure after timeout, RM:88
removing, PM:236-237
selection timeout; setting, RM:321;
value, RM:99, 112, 214
XtAddTimeout, RM:78
XtAppAddTimeOut, RM:88
XtAppSetSelectionTimeout, RM:112
XtGetSelectionTimeout, RM:99, 214
XtRemoveTimeOut, RM:304
XtSetSelectionTimeout, RM:321
XtTimerCallbackProc, RM:413
timing multiple clicks, PM:207
title option (-title), PM:93
toolkits initializing internals, RM:335
initializing toolkit and display, RM:234-238
XtInitialize, RM:234-238
XtToolkitInitialize, RM:335
top-level topLevelShellWidgetClass, RM:452-460
widget, PM:34;
(see also Shell widget.)
window, PM:548
topLevelShellWidgetClass PM:284
training PM:525
TransientShellWidgetClass PM:75
transientShellWidgetClass PM:79
TransientShellWidgetClass PM:284
transientShellWidgetClass RM:452-460
Translation Manager (see actions)
translations PM:213
augment directive, PM:50
! modifier symbol, PM:205
override directive, PM:50
replace directive, PM:50
about, PM:29, 41, 46, 49, 59, 548
augmenting, PM:50-51
colon modifier symbol, PM:205-206
compiling; table, RM:275-276;
when widget class initialized, PM:149;
XtParseTranslationTable, RM:275-276
defining; default in Core class part, PM:153;
in source, PM:51
details in, PM:198-201
differences between directives, PM:195
double-clicks, PM:206
event abbreviations, PM:196-197
event sequences, PM:206-208

hardcoding, PM:51
in widget implementation file, PM:148-149
inheritance, PM:148
interactions between, PM:208-213
keyboard events, PM:198-199
merging, PM:114, 209; widget translations, RM:119
merging/overwriting, RM:268
modifiers, PM:201-206, 203; and event sequences, PM:207
Motion events, PM:207
newlines in, PM:194
*Notify details, PM:200
order, PM:209
overriding, PM:50-51
parsing, PM:51
removing, RM:345
replacing, PM:50-51
resource converters, PM:265
syntax, PM:194
tilde modifier symbol, PM:204
translation table, PM:549; RM:711; example, PM:49
XtAugmentTranslations, RM:119
XtUninstallTranslations, RM:345
(see also accelerators; actions; XtOverrideTranslations.)
traversal post-order, PM:543
pre-order, PM:543
TrueColor PM:549
type converters PM:40
about, PM:100
explicitly invoking, PM:274-275; example, PM:274-275
format, PM:275-278
passing arguments to, PM:272; example, PM:272
registering, PM:269-274
(see also resources.)
type property PM:549
typedefs PM:142

U

unmanaging widget PM:332
UnmapNotify event RM:687-688
upward chaining PM:154
uunet PM:36
uwm PM:358

V

VendorShell widget class PM:289
Viewport widget PM:21, 110, 326
virtual colormaps PM:159, 290
crossing, RM:668
visibility interest PM:153, 237-238
VisibilityNotify event RM:701-702
visible_interest field PM:237
visual class PM:549

W

warning handler calling high-level, RM:117-118, 362-363
(see also XtAppWarningMsg; XtWarning; XtWarningMsg.)
warnings PM:399
listing, RM:703
(see also error handling.)
widget PM:15-17, 70, 105, 283
about, PM:8, 15, 549
actions; example, PM:188, 190
adding to parent list, PM:16; RM:249-251; XtManageChild, RM:249; XtManageChildren, RM:250-251
application context; getting, RM:364; XtWidgetToApplicationContext, RM:364
as data type, PM:35
call accept_focus method, RM:122
child widget; creating/managing, PM:65, 323; RM:170; layout of, PM:330-331
class; composite widget subclass, RM:246; constraint widget subclass, RM:247;

determining subclass, RM:235;
obtaining, RM:138;
Shell widget subclass, RM:234;
verifying, RM:137;
XtCheckSubclass, RM:137;
XtClass, RM:138;
XtIsComposite, RM:246;
XtIsConstraint, RM:247;
XtIsShell, RM:234;
XtIsSubclass, RM:235
composite, PM:21
converting (R3 to R4), PM:467
creating, PM:32, 85;
RM:174-176;
additional top-level widget, RM:90-91
creating additional top-level widget, RM:169
creating; working window, RM:177-178;
(see also XtAppCreateShell; XtCreateApplicationShell; XtCreateMgdWidget; XtCreateWidget.)
declaring class record pointer; example, PM:157
default size, PM:184
defining conventions, PM:138, 252;
summary, PM:164
destroying, PM:28, 84, 187; RM:182-183;
XtDestroyWidget, RM:182-183
display pointer, RM:187;
XtDisplay, RM:187
dragging, PM:20
Exclusive and Nonexclusive, PM:426
framework of code, PM:137-165
geometry (see geometry management)
getting data, PM:107;
example, PM:107
hierarchy; creating, PM:67
implementation file, PM:138, 143-160;
actions table, PM:148-149;
declaration of methods, PM:150;
resources, PM:145, 147-148;
translation table, PM:148-149
installing accelerators (see accelerators)
instance structure, PM:170
internals, PM:137-165
Label, PM:270
macros for, PM:403
management, PM:65, 184-186; RM:248;
XtIsManaged, RM:248;
(see also XtManageChild; XtManageChildren.)
managing child, PM:170-174
mapping, PM:59;
changing map_when_managed field, RM:319;
to display, RM:252
mapping to display; XtMapWidget, RM:252
mapping; windows, PM:36;
XtSetMappedWhenManaged, RM:319
merging translations, RM:119
modal widget; redirecting input, RM:73-74, 300;
XtAddGrab, RM:73-74;
XtRemoveGrab, RM:300
moving; XtMoveWidget, RM:257
moving/resizing, PM:58;
RM:140-141;
XtConfigureWidget, RM:140-141
naming conventions, PM:463
necessary include files, PM:138
parent, PM:113
parent widget; returning, RM:273;
XtParent, RM:273
popping down; Menupopdown, RM:253
popping up, PM:80-81;
Menupopup, RM:254-255
private header file, PM:138-143, 138
public header file, PM:138, 160-162
realizing, PM:180; RM:231, 287-288;
XtIsRealized, RM:231;
XtRealizeWidget, RM:287-288
record size, PM:152
removing, PM:16;
XtUnmanageChild, RM:346;
XtUnmanageChildren, RM:347

resizing, PM:69, 181-183, 322-328; RM:247, 508;
per core dimensions, RM:307;
XtMakeResizeRequest, RM:247;
XtResizeWidget, RM:306;
XtResizeWindow, RM:307
retrieving event mask, RM:120-121;
XtBuildEventMask, RM:120-121
returning screen pointer; XtScreen, RM:310
sensitivity (see sensitivity)
sets; about, PM:8, 17-18;
Motif, PM:18;
special defaults, PM:289
size; (see also widget, resizing.)
source file, PM:138
subclass; XtIsSubclass, RM:235
superclass; inheritance, PM:59;
obtaining, RM:334;
XtIsSuperclass, RM:334;
XtSuperclass, RM:334
techniques for writing, PM:163
Text, PM:433, 443
translating; coordinates, RM:336;
name to instance, RM:259;
window and display pointer, RM:367;
XtNameToWidget, RM:259;
XtTranslateCoords, RM:336;
XtWindowToWidget, RM:367
unmapping; XtUnmapWidget, RM:348
WidgetProc, RM:438
WidgetWrap, PM:98
windowless (see gadgets)
windows; destroying, RM:349;
returning, RM:365;
XtUnrealizeWidget, RM:349;
XtWindow, RM:365
(see also BitmapEdit widget; composite widget; Core widget; Scrollbar widget; Shell widget; XtCallAcceptFocus.)
widget callback list (see callbacks)
widget classes PM:65
about, PM:17
Athena, PM:19, 42, 46, 49, 65, 70, 248;
(see also Command; Label widget; Scrollbar widget; widget.)
Core (Intrinsics), RM:438-442
inheritance among (Athena), PM:19
Intrinsics, PM:16-17, 57, 61, 65, 70, 75, 139, 284, 289;
(see also applicationShell; composite widget; Core widget; OverrideShell; Shell widget; topLevelShell; TransientShell; VendorShell; widget.)
name defined, PM:152
(see also individual widgets alphabetically; widget, subclass; widget, superclass.)
widgetClass class pointer PM:120
width PM:325
checking in initialize method, PM:170
wildcards in font names, PM:454
in resource specifications, PM:39, 252
window creating widget's working window, RM:177-178
geometry, PM:458
InputOutput, RM:672
keyboard focus, RM:674
unnamed class, PM:388
viewable, PM:549
visible, PM:549
visual, PM:549
window gravity; about, PM:549
WindowObj class, PM:158
zoomed window, PM:550
(see also XtCreateWindow.)
window attributes PM:160
about, PM:158
background, PM:158
backing_store, PM:158, 269
bit_gravity, PM:158
border, PM:158
colormap, PM:159, 291
cursor, PM:159
event_mask, PM:159
override_redirect, PM:159
save_under, PM:159
setting in realize method, PM:159-160

window manager PM:69-70, 283-294, 321
about, PM:11, 33, 549
and decoration, PM:293
and icons, PM:292-293
click-to-type, PM:288
colormaps, PM:290-291
focusing styles, PM:200
hints, PM:11, 284-294
input models, PM:288
interacting with, PM:284-294
keyboard focus, PM:288-290
listener, PM:540
pointer focus, PM:288
real estate driven, PM:288, 544
redirect, PM:544
reparenting, PM:293, 373, 544
save-set, PM:546
screen layout, PM:287-288
wm_timeout RM:455
work procedures about, RM:79
creating pop up in, example, RM:420
registering, PM:239-240; RM:89
removing, PM:239; RM:305
XtAddWorkProc, RM:79
XtAppAddWorkProc, RM:89
XtAppRemoveWorkProc, RM:305
(see also XtWorkProc.)

X

X about, PM:3
X Consortium address, PM:526
X protocol, PM:5, 295
X source software, PM:521
XView, PM:9
X Toolkit (see toolkits)
XA_CLIPBOARD atom PM:303
(see also selections.)
XA_MULTIPLE property PM:316
XAPPLRESDIR environment variable PM:253
XA_PRIMARY atom PM:296, 303
(see also selections.)
XA_SECONDARY atom PM:303
(see also selections.)
XA_TARGETS atom PM:306
(see also selections.)
Xatom.h file PM:303, 305
Xaw library PM:17, 38
xbitmap application PM:106, 497
xbitmap1; example, PM:106-109
xbitmap2; example, PM:109-117
xbitmap3; example, PM:118-124
xbitmap4; example, PM:125-134
XChangeGC Xlib function PM:173
XChangeKeyboardMapping RM:690
XClearArea Xlib function PM:180
xclipboard PM:315-317
XConfigureWindow RM:661-662, 697
XConfigureWindow Xlib function PM:352
XConvertSelection RM:699
XCopyArea Xlib function PM:123, 178, 197; RM:485
XCopyColormapAndFree Xlib function PM:291
XCopyPlane Xlib function PM:123, 178, 197; RM:485
XCreateGC Xlib function PM:121, 170
XCreateWindow Xlib function PM:160
Xdefaults file PM:60
xedit PM:294
XENVIRONMENT environment variable PM:254
xev PM:203
Xext library PM:38
xfarewell.c example, PM:46-47
xfd (font displayer) PM:455
XFlush Xlib function PM:412
XGCValues structure PM:173
XGetIconSizes Xlib function PM:293
XGetModifierMapping RM:690
XGetMotionEvents RM:694
XGetPointerMapping RM:690

XGetStandardColormap Xlib function PM:291
xgoodbye.c example PM:42
xhello.c example PM:33-34
XInternAtom Xlib function PM:305-306
Xlib library PM:38, 174, 178, 180
xload PM:289
XLookupString Xlib function PM:406
XLowerWindow Xlib function PM:352
XMapRaised RM:691
XMapWindow RM:691
xmh PM:72, 294
xmodmap PM:203
XMoveResizeWindow RM:697
Xmu library PM:38, 271, 314
atom caching; initializing, PM:314
resource converters in, PM:266
XmuConvertStandardSelection, PM:314-316; RM:392
XmuInternAtom, PM:314
XParseGeometry Xlib function PM:288
XQueryPointer RM:694
XRaiseWindow Xlib function PM:352
xrdb PM:60, 251, 254
XRectInRegion Xlib function PM:177; RM:485
XRefreshKeyboardMapping RM:690
XResizeWindow RM:697
XRestackWindows Xlib function PM:352
Xrm XrmOptionDescRec; example, PM:94; structure, PM:93
XrmoptionIsArg argument style, PM:95
XrmOptionKind enum values, PM:95
XrmoptionNoArg argument style, PM:95
XrmoptionResArg argument style, PM:95
XrmoptionSepArg argument style, PM:95
XrmoptionSkipArg argument style, PM:95
XrmoptionSkipLine argument style, PM:95
XrmoptionSkipNArgs argument style, PM:95
XrmoptionStickyArg argument style, PM:95
XrmStringToQuark Xlib function, PM:276
XrmOptionDescRec RM:236
XrmoptionNoArg argument style RM:238
XSelectInput Xlib function PM:213, 222
XSendEvent RM:656, 699-700
XSetErrorHandler Xlib function PM:399
XSetInputFocus RM:675
Xlib function, PM:405
XSetIOErrorHandler Xlib function PM:399
XSetPointerMapping RM:690
XSync Xlib function PM:412
XtActionHookProc RM:373
XtActionProc PM:48; RM:374-375
XtAddActions PM:394; RM:63-64
XtAddCallback PM:44, 266; RM:65-66
arguments, PM:44
XtAddCallbacks PM:84; RM:67-68
XtAddConverter RM:69
XtAddEventHandler PM:221-222; RM:70-71
arguments, PM:224
when to call, PM:224
XtAddExposureToRegion RM:72
XtAddGrab PM:385; RM:73-74
XtAddInput PM:411; RM:75
XtAddRawEventHandler PM:227; RM:76-77
XtAddress address mode constant PM:273
XtAddressMode enum PM:273
XtAddTimeOut RM:78
XtAddWorkProc RM:79
XtAppAddActionHook PM:408, 411; RM:80

XtAppAddActions PM:47, 49, 148, 411; RM:81-82
XtAppAddConverter PM:411; RM:83-85
XtAppAddInput PM:230-232, 411; RM:86-87
XtAppAddTimeOut PM:234; RM:88
XtAppAddWorkProc PM:239; RM:89
XtAppCreateShell PM:284, 411, 413; RM:90-91
XtAppError PM:400-401, 411; RM:92
XtAppErrorMsg PM:400-401, 411; RM:93-94
XtAppGetErrorDatabase PM:411; RM:95
XtAppGetErrorDatabaseText PM:411; RM:96-98
XtAppGetSelectionTimeout PM:411
XtAppInitialize PM:32, 92, 174, 251, 254, 411; RM:100
fallback resources in, PM:257
passing options table to, PM:96
shell created, PM:79
XtAppMainLoop PM:32, 36, 143, 237, 411; RM:102
and multiple application contexts, PM:412
XtAppNextEvent PM:241, 411-412; RM:103
XtAppPeekEvent PM:411-412; RM:104-105
XtAppPending PM:411-412; RM:106
XtAppProcessEvent PM:242, 411-412; RM:107
XtAppReleaseCacheRefs PM:411; RM:108
XtAppSelectionTimeout RM:99
XtAppSetErrorHandler PM:401, 411; RM:109
XtAppSetErrorMsgHandler PM:401, 411; RM:110
XtAppSetFallbackResources PM:258, 411; RM:111
XtAppSetSelectionTimeout PM:411; RM:112
XtAppSetTypeConverter PM:411; RM:113
XtAppSetWarningHandler PM:401, 411; RM:114
XtAppSetWarningMsgHandler PM:401, 411; RM:115
XtAppTimeOut PM:411
XtAppWarning PM:400-401, 411; RM:116
XtAppWarningMsg PM:400-401, 411; RM:117-118
XtAppWorkProc PM:411
XtArgsProc RM:499
XtArgVal PM:99
XtAugmentTranslations PM:51, 115; RM:119
reasons to use, PM:114
XtBaseOffset address mode constant PM:273
XtBuildEventMask RM:120-121
XtCallAcceptFocus PM:157, 405; RM:122
XtCallActionProc PM:408; RM:123
XtCallbackExclusive PM:84, 365; RM:124-125
XtCallbackList RM:376
XtCallbackNone PM:84, 365; RM:126
XtCallbackNonexclusive PM:84, 365; RM:127
XtCallbackPopdown PM:365; RM:128-129
XtCallbackProc RM:376-377
XtCallbackRec RM:376
format, PM:85
XtCallbackReleaseCacheRef RM:130
XtCallbackReleaseCacheRefList RM:131
XtCallCallbackList RM:132
XtCallCallbacks PM:41; RM:133
XtCallConverter PM:271, 275; RM:134
XtCalloc PM:187, 408; RM:136
XtCancelConvertSelectionProc PM:317; RM:378
XtCaseProc RM:379-380
function prototype, PM:407
XtCheckSubclass RM:137

XtClass RM:138
XtCloseDisplay RM:139
XtConfigureWidget PM:58; RM:140-141
XtConvert PM:274; RM:142-143
XtConvertAndStore PM:100, 271, 274-275; RM:164-166
XtConvertArgProc RM:381-382
XtConvertArgRec PM:272
XtConvertCase PM:407; RM:167
XtConverter RM:383-386
XtConvertSelectionIncrProc PM:317; RM:387-388
XtConvertSelectionProc PM:296, 304; RM:389-392
XtCreateApplicationContext RM:168
XtCreateApplicationShell PM:411, 413; RM:169
XtCreateManagedWidget PM:32, 68, 101, 142, 251; RM:170, 474
XtCreatePopupChildProc PM:384
XtCreatePopupShell PM:79, 284; RM:171-173
XtCreateWidget PM:32, 142, 169, 251; RM:174-176, 436
  hardcoding resources with, PM:99
XtCreateWindow PM:160; RM:177-178
XtDatabase RM:179
XtDefaultBackground PM:265, 277, 290
XtDefaultFont constant PM:265
XtDefaultForeground constant PM:265, 277, 290
XtDestroyApplicationContext RM:180
XtDestroyGC RM:181
XtDestroyWidget PM:169, 335; RM:182-183, 436, 500
XtDestructor RM:393
XtDirectConvert RM:184
XtDisownSelection PM:312, 318; RM:185
XtDispatchEvent PM:241; RM:186
XtDisplay PM:275; RM:187
XtDisplayInitialize RM:188-189
XtDisplayOfObject PM:404; RM:190
XtDisplayStringConversionWarning PM:276-277, 402; RM:191
XtDisplayToApplicationContext PM:412; RM:192
xterm PM:288, 358
XtError PM:411; RM:193
XtErrorHandler RM:394
XtErrorMsg PM:411; RM:194-195
XtErrorMsgHandler RM:395-396
xtetris PM:234-236
XtEventHandler RM:397-398
XtFilePredicate RM:399
XtFindFile PM:409; RM:196
XtFree PM:187, 311, 408; RM:198
XtGeometryAlmost PM:186
XtGeometryMask RM:245, 285, 488, 502, 515
XtGeometryNo PM:186
XtGeometryResult PM:185, 332; RM:245, 247, 285
XtGeometryYes PM:186
XtGetActionKeysym PM:407; RM:199
XtGetApplicationNameAndClass PM:404; RM:200
XtGetApplicationResources PM:85, 252; RM:201-203
XtGetConstraintResourceList PM:338; RM:204
XtGetErrorDatabase PM:401, 411; RM:205
XtGetErrorDatabaseText PM:401, 411; RM:206
XtGetGC PM:121, 170, 173, 180; RM:207-208
  when not to use, PM:173
XtGetKeysymTable PM:407; RM:209
XtGetMultiClickTime RM:210
XtGetResourceList PM:148, 263; RM:211-212
XtGetSelectionRequest PM:316; RM:213

XtGetSelectionTimeout PM:318, 411; RM:214
XtGetSelectionValue RM:215
XtGetSelectionValueIncremental PM:317; RM:216
XtGetSelectionValues PM:296, 304-306, 313, 318; RM:218-219
XtGetSelectionValuesIncremental RM:220
XtGetSubresources PM:279; RM:221-224
XtGetSubvalues PM:280; RM:223-224
XtGetValues PM:24, 57, 80, 373; RM:225-226, 436
XtGrabButton PM:365, 409; RM:227-228
XtGrabKey PM:365, 409; RM:229-230
XtGrabKeyboard PM:409; RM:231
XtGrabNone grab mode PM:384
XtGrabPointer PM:364, 409; RM:232
XtHasCallbacks PM:404; RM:233
XtIMAlternateInput constant PM:242
XtImmediate address mode constant PM:273
XtIMTimer constant PM:242
XtIMXEvent constant PM:242
XtInherit constants, PM:154;
XtInheritDeleteChild, PM:335;
XtInheritGeometryManager, PM:333;
XtInheritInsertChild, PM:335;
XtInheritRealize, PM:158;
XtInheritSetValuesAlmost, PM:334
XtInheritInsertChild RM:499
XtInitialize RM:234-238
XtInitializeWidgetClass RM:239
XtInputCallbackProc RM:400-401
XtInsertEventHandler PM:224; RM:240
arguments, PM:224
when to call, PM:224
XtInsertRawEventHandler RM:241
XtInstallAccelerators PM:211; RM:242-243
XtInstallAllAccelerators PM:211, 215; RM:244
XtInternAtom Xlib function PM:303
XtIsApplicationShell RM:245
XtIsComposite PM:403; RM:246
XtIsConstraint RM:247
XtIsManaged PM:403; RM:248
XtIsObject RM:229
XtIsOverrideShell RM:230
XtIsRealized PM:176, 403-404; RM:231, 484
XtIsRectObj RM:232
XtIsSensitive PM:404; RM:233
XtIsShell RM:234
XtIsSubclass PM:403; RM:235
XtIsTopLevelShell RM:236
XtIsTransientShell RM:237
XtIsVendorShell RM:238
XtIsWidget RM:239
XtIsWMShell RM:240
XtKeyProc RM:402-404
function prototype, PM:407
XtKeysymToKeycodeList PM:407; RM:241
XtLastTimestampProcessed RM:242
XtLoseSelectionIncrProc PM:317; RM:405
XtLoseSelectionProc PM:304; RM:406
XtMainLoop PM:411; RM:243
XtMakeGeometryRequest PM:333; RM:244-246
XtMakeResizeRequest RM:247
XtMalloc PM:187, 408; RM:248
XtManageChild PM:35; RM:249
XtManageChildren PM:32, 35; RM:250-251
XtMapWidget PM:332; RM:252
XtMenuPopdown RM:253
XtMenuPopup PM:369; RM:254-255
argument to, PM:370
XtMergeArgLists PM:102; RM:256

XtMoveWidget PM:58, 324, 331; RM:257
XtN constants PM:40
XtName RM:258
XtNameToWidget PM:404
XtNbasewidth (Shell) PM:286
XtNcolormap setting, PM:291
XtNcreatePopupChildProc resource PM:384
XtNew PM:408; RM:260, 517
XtNewString PM:408; RM:261
XtNextEvent PM:411; RM:262
XtNinitialResourcesPersistent PM:271
XtNinput setting, PM:290
XtNmultiClickTime resource PM:207
XtNsensitive resource PM:81
XtNumber RM:263
XtOffset PM:88; RM:264
XtOffsetOf macro PM:251; RM:265
XtOpenDisplay PM:413; RM:266-267
XtOrderProc prototype procedure RM:407
XtOverrideTranslations PM:51, 114-115; RM:268
reasons to use, PM:114
XtOwnSelection PM:296, 303-304; RM:269-270
XtOwnSelectionIncremental PM:317; RM:271-272
XtParent RM:273
XtParseAcceleratorTable PM:216; RM:274
XtParseTranslationTable PM:51, 115, 149, 216; RM:275-276
XtPeekEvent PM:241, 411; RM:277
XtPending PM:241, 411; RM:278
XtPointer PM:268, 389
XtPopdown PM:75, 365, 369; RM:279, 460
XtPopup PM:75, 365, 377; RM:280-281, 460
XtPopupSpringLoaded PM:365; RM:282
XtProc RM:438
XtProcedureArg address mode constant PM:273
XtProcessEvent PM:411; RM:283
XtQueryGeometry PM:325; RM:284-286
XtRAcceleratorTable representation type PM:265
XTranslateCoordinates Xlib function PM:373
XtRBackingStore representation type PM:269
XtRBitmap representation type PM:269
XtRBool representation type PM:265
XtRBoolean representation type PM:265
XtRCallback representation type PM:270
XtRCallProc representation type PM:251, 268
XtRCursor representation type PM:265, 269
XtRDimension representation type PM:265
XtRDisplay representation type PM:265
XtRealizeWidget PM:32, 35, 156, 324, 332; RM:287-288
XtRealloc PM:408; RM:289
XtRegisterCaseConverter PM:407; RM:290
XtRegisterGrabAction PM:365; RM:291-292
XtReleaseGC PM:180; RM:293
XtRemoveAccelerators PM:212
XtRemoveActionHook PM:408; RM:294
XtRemoveActions PM:148
XtRemoveAllActions PM:148
XtRemoveAllCallbacks RM:295
XtRemoveCallback PM:84, 372; RM:296
XtRemoveCallbacks PM:84; RM:297
XtRemoveEventHandler PM:187, 226; RM:298-299
XtRemoveGrab PM:385; RM:300
XtRemoveInput PM:232; RM:301

XtRemoveRawEventHandler PM:227; RM:302-303
XtRemoveTimeOut PM:187, 236; RM:304
XtRemoveWorkProc PM:239; RM:305, 419
XtResizeWidget PM:324, 331; RM:306
XtResizeWindow RM:307
XtResolvePathname PM:409; RM:308-309
XtResource PM:248
XtResourceDefaultProc RM:408
example, PM:268
XtResourceQuark address mode constant PM:273
XtResourceString address mode constant PM:273
XtRFile representation type PM:265
XtRFloat representation type PM:265
XtRFont representation type PM:265
XtRFontStruct representation type PM:265
XtRFunction representation type PM:270
XtRGeometry representation type PM:265
XtRImmediate representation type PM:251, 267
XtRInitialState representation type PM:265
XtRInt representation type PM:265
XtRJustify representation type PM:269
XtROrientation representation type PM:269
XtRPixel representation type PM:265
XtRPosition representation type PM:265
XtRShort representation type PM:265
XtRTranslationTable representation type PM:265
XtRUnsignedChar representation type PM:265
XtRVisual representation type PM:265
XtRWidget representation type PM:269
XtScreen RM:310
XtScreenOfObject PM:404; RM:311
XtSelectionCallbackProc PM:296, 317; RM:409-410
XtSelectionDoneIncrProc PM:317; RM:411
XtSelectionDoneProc PM:296; RM:412
XtSetArg PM:55, 57, 100, 121; RM:312-313
XtSetErrorHandler PM:411; RM:314
XtSetErrorMsgHandler PM:411; RM:315
XtSetKeyboardFocus PM:405; RM:316-317
XtSetKeyTranslator RM:318
XtSetMappedWhenManaged RM:319
XtSetMultiClickTime RM:320
XtSetSelectionTimeout PM:318, 411; RM:321
XtSetSensitive PM:81, 384; RM:322
XtSetSubvalues PM:280; RM:323-324, 517
XtSetTypeConverter PM:271-272; RM:325
arguments, PM:270
XtSetValues PM:24, 51, 55, 156, 159, 331, 349; RM:327-328, 436
XtSetValuesFunc PM:180
XtSetWarningHandler PM:411; RM:329
XtSetWarningMsgHandler PM:411; RM:331
XtSetWMColormapWindows RM:332
XtShellExtensionVersion RM:455
XtStringConversionWarning RM:333
XtSuperclass RM:334
XtTimeOut PM:411
XtTimerCallbackProc RM:413
XtToolkitInitialize RM:335

XtTranslateCoords PM:80, 372-373; RM:336
XtTranslateKey PM:407; RM:337-338
XtTranslateKeycode PM:407; RM:339-340
XtTypeConverter RM:414-418
XtUngrabButton PM:409; RM:341
XtUngrabKey PM:409; RM:342
XtUngrabKeyboard PM:409; RM:343
XtUngrabPointer PM:364, 409; RM:344
XtUninstallTranslations RM:345
XtUnmanageChild PM:332; RM:346
XtUnmanageChildren PM:332; RM:347
XtUnmapWidget PM:332; RM:348
XtUnrealizeWidget RM:349
XtVaAppCreateShell PM:413; RM:350
XtVaAppInitialize PM:32, 34; RM:351
arguments, PM:35
fallback resources in, PM:257
shell created, PM:79
XtVaCreateArgsList RM:352
XtVaCreateManagedWidget PM:32, 35, 142, 251; RM:353
XtVaCreatePopupShell PM:79, 360; RM:354
XtVaCreateWidget PM:32, 67, 142, 251; RM:355
XtVaGetApplicationResources RM:356
XtVaGetSubresources PM:279; RM:357
XtVaGetSubvalues PM:280; RM:358
XtVaGetValues PM:53, 138; RM:359
XtVaSetSubvalues PM:280; RM:360
XtVaSetValues PM:53, 138, 142, 169-170, 178-181; RM:361
XtVersion constant PM:153
XtVersionDontCheck constant PM:153
XtWarning PM:411; RM:362
XtWarningMsg PM:411; RM:363
XtWidgetBaseOffset address mode constant PM:273
XtWidgetGeometry PM:332, 335
XtWidgetGeometry structure PM:184-185
XtWidgetToApplicationContext PM:236; RM:364
XtWindow PM:180, 391; RM:365
XtWindowOfObject PM:404; RM:366
XtWindowToWidget PM:404; RM:367
XtWorkProc PM:411; RM:419-420

O'Reilly & Associates, Inc.

Creators and Publishers of Nutshell Handbooks,
concise, down-to-earth guides on selected UNIX topics

The X Window System series:

Vol. 0 *X Protocol Reference Manual*
Vol. 1 *Xlib Programming Manual*
Vol. 2 *Xlib Reference Manual*
Vol. 3 *X Window User's Guide*
Vol. 4 *X Toolkit Intrinsics Progamming Manual*
Vol. 5 *X Toolkit Intrinsics Reference Manual*
Vol. 7 *XView Programming Manual*

and *The X Window System in a Nutshell,*
a quick reference

Send me more information on:

❑ O'Reilly catalog and newsletter
❑ Placing a standing order for new titles
❑ Retail sales
❑ Corporate sales
❑ Bookstore locations
❑ Overseas distributors
❑ Upcoming books on the subject:

_____

_____

❑ Writing a Nutshell Handbook

O'Reilly & Associates, Inc.

Creators and Publishers of Nutshell Handbooks,
concise, down-to-earth guides on selected UNIX topics

The X Window System series:

Vol. 0 *X Protocol Reference Manual*
Vol. 1 *Xlib Programming Manual*
Vol. 2 *Xlib Reference Manual*
Vol. 3 *X Window User's Guide*
Vol. 4 *X Toolkit Intrinsics Progamming Manual*
Vol. 5 *X Toolkit Intrinsics Reference Manual*
Vol. 7 *XView Programming Manual*

and *The X Window System in a Nutshell,*
a quick reference

Send me more information on:

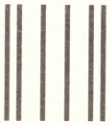
❑ O'Reilly catalog and newsletter
❑ Placing a standing order for new titles
❑ Retail sales
❑ Corporate sales
❑ Bookstore locations
❑ Overseas distributors
❑ Upcoming books on the subject:

_____

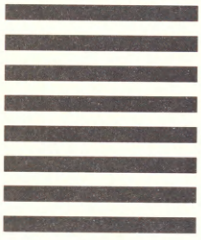
_____

❑ Writing a Nutshell Handbook

NAME _____
COMPANY _____
ADDRESS _____
CITY _____ STATE _____ ZIP _____



NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 80 SEBASTOPOL, CA

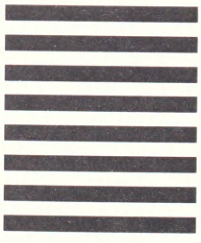
POSTAGE WILL BE PAID BY ADDRESSEE

O'Reilly & Associates, Inc.

632 Petaluma Avenue
Sebastopol, CA 95472-9902

NAME _____
COMPANY _____
ADDRESS _____
CITY _____ STATE _____ ZIP _____

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 80 SEBASTOPOL, CA

POSTAGE WILL BE PAID BY ADDRESSEE

O'Reilly & Associates, Inc.

632 Petaluma Avenue
Sebastopol, CA 95472-9902

Overseas Distributors

Effective January 1, 1990, customers outside the U.S. and Canada will be able to order Nutshell Handbooks, the Pick Series, and the X Window System Series through distributors near them. These overseas locations offer international customers faster order processing, more local bookstores and local distributors, and increased representation at trade shows worldwide, as well as the high level, quality service our customers have always received.

AUSTRALIA & NEW ZEALAND
(orders and inquiries)
Addison-Wesley Publishers, Pty. Ltd.
6 Byfield Street
North Ryde, N.S.W. 2113
AUSTRALIA
Telephone: 61-2-888-2733
FAX: 61-2-888-9404

UNITED KINGDOM & AFRICA
(orders and inquiries)
Addison-Wesley Publishers, Ltd.
Finchampstead Road
Wokingham, Berkshire RG11 2NZ
ENGLAND
Telephone: 44-734-794-000
FAX: 44-734-794-035

EUROPE & THE MIDDLE EAST
(orders and inquiries)
Addison-Wesley Publishers B.V.
De Lairessestraat 90
1071 PJ Amsterdam
THE NETHERLANDS
Telephone: 31-20-764-044
FAX: 31-20-664-5334

ASIA inquiries (excluding Japan)
Addison-Wesley Singapore Pte. Ltd.
15 Beach Road #05-09/10
Beach Centre
Singapore 0718
SINGAPORE
Telephone: 65-339-7503
FAX: 65-339-9709

ASIA orders (excluding Japan)
Addison-Wesley Publishing Co., Inc.
International Order Department
Route 128
Reading, Massachusetts 01867 U.S.A.
Telephone: 1-617-944-3700
FAX: 1-617-942-2829

JAPAN
(orders and inquiries)
Toppan Company, Ltd.
Ochanomizu Square B, 1-6
Kanda Surugadai
Chiyoda-ku, Tokyo 101
JAPAN
Telephone: 81-3-295-3461
FAX: 81-3-293-5963

Maruzen Company, Ltd.
3-10 Nihonbashi 2-Chome
Chuo-Ku, Tokyo, 103
JAPAN
Telephone: 81-3-272-7211
FAX: 81-3-274-3238

X Toolkit Intrinsics Programming Manual

Volume 4, *X Toolkit Intrinsics Programming Manual*, is a complete programmer's guide to the X Toolkit (Xt). Xt is a library of the C language routines designed to facilitate the implementation of user interfaces, with reusable components called widgets. This book uses the Athena widgets to demonstrate how to use existing widgets, but it is equally applicable to and provides a good introduction to programming with any other widget set based on Xt, such as OSF/Motif or AT&T's OPEN LOOK widget set. It includes chapters on:

- Introduction to the X Window System
- Building applications with widgets
- A bitmap editor constructed with widgets
- Inside a widget
- Basic widget methods
- Events, translations and accelerators
- Event handlers, timeouts and work procedures
- Resource management and type conversion
- Selections and window interaction manager
- Geometry management
- Menus, gadgets and cascaded pop ups
- Miscellaneous Toolkit programming techniques
- Comparison of Athena, OSF/Motif and AT&T's OPEN LOOK widgets

as well as a glossary, a master index and many useful appendices.

This book is designed to be used with the Volume Five, *X Toolkit Intrinsics Reference Manual*, which provides reference pages for each of the Xt functions and the widget classes defined by Xt.

Volume 4 ISBN: 0-937175-56-0
Volumes 4 and 5 (set) ISBN: 0-937175-58-7

O'Reilly & Associates, Inc.